Tomi Harju

Angular ja Grails web-sovelluskehityksessä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ohjelmistotekniikka

Insinöörityö

28.2.2015

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Tomi Harju Angular ja Grails web-sovelluskehityksessä 39 sivua + 1 Liitettä 28.2.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Projektipäällikkö Mikko Kaasinen

Työn tavoitteena oli toteuttaa Angularia ja Grailsia hyödyntävä projektinhallintasovellus sekä selittää näiden kahden sovelluskehyksen perusteet kyseistä sovellusta esimerkkinä käyttäen. Työn lukemisen jälkeen lukijan on helpompi aloittaa sovelluskehyksien tarkempi opettelu. Projekti tehtiin varsinaisen päivätyön ohella Digia Finland Oy:lle, ainakin aluksi vain yrityksen sisäiseen käyttöön.

Teksti koostuu kolmesta kokonaisuudesta. Ensimmäisessä perehdytään Grailsiin, toisessa Angulariin ja kolmannessa paneudutaan esimerkkisovelluksen avulla näiden kahden yhteistyöhön sekä kuvataan, kuinka esimerkkisovellus toteutettiin. Työssä pyrittiin kertomaan tärkeimmistä ja keskeisimmistä ominaisuuksista. Silti useisiin tärkeisiin toimintoihin viitataan vain ohimennen, jolloin lukijan täytyy tutkia niitä tarkemmin erikseen.

Työn lopputuloksena syntyi toimiva prototyyppi, joka jo sellaisenaan on hyödyllinen projektinseurantaan. Sovelluksen kehitystyö ei rajoitu tämän työn piiriin, vaan jatkokehitys alkanee pian tämän työn valmistumisen jälkeen.

Avainsanat	Grails, Angular, Groovy, Spring, JavaScript, API, Single-page-app

Author(s) Title Number of Pages Date	Tomi Harju Angular and Grails in Web Development 39 pages + 1 appendices 28.2.2015
Degree	Bachelor of Engineering
Degree Programme	Information technology
Specialisation option	Software engineering
Instructor(s)	Simo Silander, Lecturer Mikko Kaasinen, Project Manager

The aim of this thesis is to produce a simple prototype of a project management tool, and to teach the basics of Angular and Grails using that tool as an example. Having read this thesis, it is easier for the reader to start working with these two web-development frameworks. The project was implemented for Digia corporation's internal use alongside daytime work.

Text consists of three major parts. The first one is about Grails, the second about Angular and the third is about the collaboration of these two. In each section, the examples are from the actual project. Text tries to cover all the main features from Angular and Grails, but some functions are only described in brief and thus it is up for the reader to figure those out.

As a result, a working prototype was developed which already has some useful features for project management. Development of the tool is not restricted to this thesis, further development will be conducted later.

Keywords	Grails, Angular, Groovy, Spring, JavaScript, API, Single-page-app

Sisällys

Lyhenteet

GORM	Grails object relation mapping, Grailsin käyttämä relaatiomalli, jonka avulla tietokantaa voidaan käsitellä olioiden avulla.
XML	Extensible markup language, merkintäkieli, jolla voi tehdä sekä ihmis- että koneluettavaa sisältöä.
JSON	JavaScript object notation, kevyt ja helposti luettava tiedonvälitysstandardi, jolla välitetään tietoa palvelimen ja asiakassovelluksen välillä.
JavaScript	Yleisimmin web-selaimissa ajettava, dynaamisesti tyypitetty skriptikieli.
CSS	Cascading style sheets, määrittää jollain kuvauskielellä luodun dokumentin ulkoasun.
CDN	Content delivery network, suuri hajautettu palvelinverkko, joka tarjoaa luotettavasti eri sisältöä käyttäjille.
GVM	Groovy Environment Manager, työkalu, jonka avulla esimerkiksi Grailsin versiomuutosten hallinta on helppoa.
MVC	Model view controller, sovellusten kolmiosainen arkkitehtuurimalli.
DDD	Domain driven design, asia- tai ominaisuuslähtöinen suunnittelumalli.
API	Application programming interface, ohjelmointirajapinta, jonka avulla ohjelmat voivat kommunikoida keskenään.
SPA	Single page app, web-sivu tai -sovellus, joka toimii yhdellä sivulla.
Backend	Sovelluksen taustajärjestelmä, joka vastaa logiikasta ja tiedon tallentamisesta.
Frontend	Sovelluksen käyttäjärajapinta, jonka avulla loppukäyttäjä käyttää sovellusta.

AJAX	Asynchronous JavaScript and XML, tekniikka, jolla web-sovellukset voivat tehdä asynkronisia kutsuja palvelimelle.
TDD	Test driven development, sovelluskehitysmalli, jossa testit kirjoitetaan ennen varsinaista toiminnallisuutta.
URL	Uniform resource locator, nimi tai osoite, joka viittaa johonkin internetissä olevaan resurssiin, yleisimmin www-sivuun.

1 Johdanto

Nykypäivänä web-sovellusta suunniteltaessa törmää valinnan vaikeuteen, ja markkinat ovat täynnä ilmaisia sovelluskehyksiä. Useat sisältävät vain jonkin tietyn sovelluksen osan kuten loppukäyttäjän sovelluksen tai tietokantasovelluksen. Olemassa on myös ns. Full Stack -kehyksiä, jotka tarjoavat ratkaisut tietokantatasolta loppukäyttäjän sovellukseen asti. Työkaluja valittaessa tulee ottaa huomioon ainakin sovelluksen tarpeet, tietokannan vaatimukset sekä kehittäjien osaaminen. Jokaisen uuden teknologian opettelu vaatii aina resursseja, mikä hidastaa tuotteen valmistumista ja lisää kustannuksia. Ei siis ole suositeltavaa aloittaa uutta suurta projektia täysin tuntemattomalla teknologialla. Siirtyminen uuteen kannattaa tehdä pienissä osissa aloittaen sovelluksen pienimmistä ominaisuuksista. Kehitys on ollut ja tulee olemaan niin nopeaa, että ”sen parhaan” sovelluskehyksen odottaminen on turhaa. Tänään aloitetun projektin olisi varmasti voinut tehdä paremmin kuukauden päästä julkaistavalla uudella kehyksellä. Sen vuoksi kehittäjien tulee sitoutua valittuun teknologiaan tietyn projektin osalta, seuraavassa projektissa tekniikkaa voidaan taas vaihtaa. Toinen toimiva idea on useiden pienempien kehysmoduulien hyödyntäminen, jolloin jokin pieni sovelluksen osa on helpompi vaihtaa uuteen.

Tässä työssä perehdytään kahteen uudehkoon web-sovelluskehykseen, Grailsiin sekä Angulariin. Grails on edellä mainittu Full stack -kehys; se tarjoaa toiminnot tietokannasta loppukäyttäjän sovellukseen asti. Angular on selaimessa toimiva Full stack -kehys, vaikka siinä ei ole tietokantaominaisuuksia. Molemmat toimivat MVC-mallin mukaan, Grailsissa malli (Model) on tietokannassa oleva objekti, kun taas Angularissa mallit ovat selaimen muistissa olevia muuttujia, jotka ovat peräisin joko palvelimen tietokannasta tai ohjelman sisäisistä tiedoista. Kummatkin teknologiat ovat lähes monoliittisia, eli niiden osia on vaikea irrottaa toisistaan ja vaihtaa. Mutta ainakin Angularin näkymät (View) voidaan piirtää käyttäen Facebookin kehittämää React.js-kirjastoa, jolla voidaan saavuttaa suuriakin suorituskykyparannuksia.

Työn tavoitteena on luoda perusteet kattava opas, jonka avulla lukija pääsee hyvin alkuun näiden uusien teknologioiden käyttöönotossa. Työssä tutustutaan ensin Grailsiin, käydään läpi Grails-sovelluksen perusrakenne, sen osat ja toimintaperiaatteet. Tämän jälkeen tutustutaan Angulariin ja sen rakenteeseen. Molempien ollessa MVC-mallin mukaisia sovelluskehyksiä mahdolliset yhteneväisyydet pyritään tuomaan ilmi. Sanallisen opastuksen lisäksi asioita selvennetään koodiesimerkein sekä kuvaajin. Joitakin sovellusten kannalta tärkeitä-

kin asioita mainitaan vain ohimennen, mutta jokaisen tällaisen asian yhteyteen lisätään lähde, josta lisätietoa voi halutessaan saada.

Viimeisessä osiossa ensimmäisten lukujen asiat kootaan yhteen ja sovelletaan Digia Finland Oy:lle tehdyn ohjelmistoprojektin kontekstissa. Projekti käydään läpi kohta kohdalta aina suunnittelusta prototyypin valmistumiseen asti. Osion tärkein tavoite on esittää yksi toimivaksi todettu tapa, jolla Angularin ja Grailsin välinen tehokas yhteistyö on mahdollista toteuttaa. Projektin avulla on tarkoitus pilotoida Angularin käyttöä olemassa olevan Grails-sovelluksen kanssa. Mikäli kokeilu onnistuu, koko sovelluksen frontend toteutetaan uudelleen käyttäen Angularia.

2 Projektin määrittely

2.1 Ongelman kuvaus

Projektinhallinta tuo useita haasteita niin pienille kuin suurillekin yrityksille. Projekteissa on monia osapuolia, joista jokaisella on omat roolinsa, tehtävänsä sekä ennemmin tai myöhemmin myös tottumuksensa. Usein yrityksillä on käytössä erinäisiä malleja, joilla projektin vaiheita pyritään ohjaamaan, mutta mikään tai kukaan ei valvo näiden mallien toteutumista, vaan se on jokaisen projektissa mukana olevan omalla vastuulla. Useasti kiireestä, huolimattomuudesta tai muusta syystä johtuen nämä ohjesäännöt tai mallit sivuutetaan kokonaan. Pahimmassa tapauksessa projektin eri vaiheiden dokumentaatiot ovat hajallaan työntekijöiden omilla kovalevyillä, sähköposteissa tai tulosteina salkkujen pohjilla. Hajallaan oleva dokumentointi vaikeuttaa tiedon löytymistä ja täten aikaisemmin sovittuihin asioihin on vaikeampi palata. Kuvitellaan, että projektin määrittelyvaiheessa on aikoinaan laskettu sen tuottavan 25 % katteen, mutta projektin loppusuoralla kate onkin jo tappiollinen. Tällaisessa tilanteessa pitäisi pystyä palaamaan myyntivaiheessa laskettuihin ja sovittuihin yksityiskohtiin ja kyetä selvittämään, mikä laskelmissa alun perin meni pieleen. Jos alun dokumentaatioita ei ole tehty yrityksen mallien mukaisesti, tai ne ovat muuten vaan hukkuneet, ei voida koskaan analysoida, mikä projektissa meni väärin.

2.2 Projektin tavoite

Tavoitteena on suunnitella ja toteuttaa prototyyppi nopeasta ja helppokäyttöisestä projektien eri vaiheiden seurantaa helpottavasta työkalusta. Työkalu liitetään jo olemassa olevaan sovellukseen erillisenä liitännäisenä, jolloin sen käyttöönotto helpottuu ja perustoiminnot kuten käyttäjänhallinta ovat jo valmiina.

Prototyypin avulla käyttäjän tulisi voida luoda uusia projekteja, lisätä niihin vaiheita sekä luoda vaiheisiin omia muistilistoja. Muistilistan kohteita pitää pystyä merkitsemään tehdyiksi sekä kommentoimaan. Tärkeimpänä ominaisuutena jokaisen kohdan (projektin, vaiheen, muistilistan) pystyy tallentamaan sapluunaksi uusia projekteja varten.

2.3 Käytössä olevat työkalut

Sovellus toteutetaan Pivotalin Grails-version 2.3.11 sekä Googlen AngularJS-version 1.3.3 avulla. Ohjelma, jonka liitännäiseksi työkalu tehdään, käyttää Grailsia sekä backendin että frontendin puolella. Grails kuitenkin mahdollistaa, että osa sovelluksesta käyttää sitä vain pilvipalveluna, jolloin loppukäyttäjän puoli voidaan toteuttaa Angularilla. Voidaan siis ajatella, että työkalu toteutetaan ohjelmaksi ohjelman sisään, työkalun teknisestä kuvauksesta lisää myöhemmin. Teoriassa Grailsia voi kehittää millä tahansa tekstieditorilla, mutta tässä työssä käytetään maksullista IntelliJ:n IDEA editoria, sen sisältämän hyvän Grails-tuen ja muidenkin ominaisuuksien kuten virheenjäljityksen ja kattavien hakutoimintojen takia. Toinen mahdollinen vaihtoehto on ilmainen Eclipse STS, joka omien kokemusten perusteella on hieman kankea ja hidas. Pilvipalvelun toteuttamiseen käytetään palvelinta, jolle on asennettuna Apache, MySQL sekä Grailsin suorittamiseen vaadittava Apache Tomcat. Kehitystyö tehdään Oraclen Virtualbox:issa toimivalla Ubuntu-käyttöjärjestelmällä, versionhallinta GIT:llä. Näiden lisäksi sovelluksen toiminta edellyttää useiden liitännäisten ja muiden aputyökalujen käyttöä, joista kerrotaan kunkin käyttötilanteen yhteydessä.

3 Grails

3.1 Yleinen esittely

Grails, aijemmin tunnettu nimellä ”Groovy on Rails”, on vuonna 2006 ilmestynyt täysialainen web-sovelluskehys (full-stack). Käytännössä Grails-sovellus on Spring MVC -sovellus. jonka päälle on lisätty ohjelmointia helpottava ja konfiguroinnin tarvetta vähentävä kerros. Grailsia ohjelmoidaan Java-pohjaisella Groovy-nimisellä ohjelmointikielellä. [1.] Groovy siis tukee täysin Javan syntaksia, mutta se sisältää useita ohjelmointia nopeuttavia ja helpottavia ominaisuuksia. Sen sanotaan olevankin ”Java on steroids”. Käytännössä Groovy muistuttaa enemmän JavaScriptiä kuin Javaa. Groovya ajetaan Javan virtuaalikoneessa, mikä erottaakin Grailsin muista kehyksistä kuten Ruby:sta tai PHP:sta, jotka eivät virtuaalikonetta tarvitse. Suorittaakseen Grails-ohjelman kehittäjä tarvitsee Javan kehitystyökalut (Java Development Kit) sekä Grails-asennuksen.

Grails on täysin ilmainen. Sen asentaminen ja eri versioiden hallinta Ubuntu-ympäristössä onnistuu helpoiten GVM-työkalun avulla. [2.] Grails-sovellus käyttää MVC-arkkitehtuuria, jossa tietokantamallit, ohjaimet ja näkymät on eroteltu toisistaan. Grailsillä kehittäminen on nopeaa ja helppoa, sillä täysin toimivan sovelluksen voi tehdä ilman minkäänlaista erillistä konfigurointia. Oppimiskäyrä on loiva, dokumentaatio on hyvin kattava ja sisältää paljon esimerkkejä. Täten tässä työssä sivuutetaankin useimpien perustoimintojen kuten sovelluksen luomiseen ja käynnistämiseen liittyvät toimenpiteet. Käytön yleistyttyä apua saa hyvin myös internetin eri yhteisöistä. Toisaalta vielä vuodenkin kokemuksen jälkeen siitä löytyy uusia ominaisuuksia. Sovelluksen kehityksessä suositaan Domain Driven Designia (DDD), joka tarkoittaa, että kehitys aloitetaan luomalla yksittäisiä tietoyksiköitä, joiden yhteyteen lisätään logiikkaa. Seuraavassa luvussa perehdytään tarkemmin Grailsin toimintaan; perehtyminen aloitetaan tietoyksiköiden eli domainien luomisella.

3.2 Grails-sovelluksen rakenne

3.2.1 Tietokantamallinnus

Lähes kaikki internetsivut sisältävät enemmän tai vähemmän tietoa, jota halutaan lisätä, muokata tai poistaa. Tietokannat ovat siis kehityksen tärkein lähtökohta, ja toteutus aloitetaankin useimmiten tietokantamallien suunnittelulla.

Grailsissa on Hibernaten päälle rakennettu relaatiomalli ”GORM” (Grails object relation mapping), jonka avulla tietokantamallinnus tehdään. Relaatiomallit mahdollistavat oliolähtöisen tietokantasuunnittelun, eikä kehittäjän juuri tarvitse miettiä tietokannan lopullista taulurakennetta. [3.] Oliomallinnus toteutetaan tavallisten POGO (Plain Old Groovy Object, vrt. POJO) -luokkien avulla. Luokat sisältävät erilaisia muuttujia ja sääntöjä joiden mukaan GORM luo lopulliset tietokantataulut ohjelman suorituksen yhteydessä tai erillisen tietokannan päivityskomennon aikana. Grails hyödyntää ”coding by convention” -ajatusmallia, mikä esimerkiksi domain-luokkien osalta tarkoittaa sitä, että taulun nimeksi tulee itse Groovy-luokan nimi. Yleisesti ottaen se tarkoittaa sitä, että liiallisen konfiguroinnin sijaan hyödynnetään ennalta sovittuja käytäntöjä, jotka on alun jälkeen helppo sisäistää. [4.]

Kuvista 1 ja 2 nähdään, kuinka ”authority”-niminen merkkijonomuuttuja muutetaan tietokantaan samannimiseksi kentäksi. Lisäksi luokan nimi ”Role” on muutettu sopimusten mukaisesti tietokantatauluksi nimeltä ”role”. Muuttujien lisäksi GORM lisää kaksi muutakin kenttää: id- sekä versiokentän. Id on tietueen yksilöivä numero, jonka arvo kasvaa aina, kun uusi tietue luodaan. Versio on myös Grailsin ylläpitämä muuttuja, joka pitää nimensä mukaan kirjaa tietueen versiosta. Sitä voidaan hyödyntää tilanteissa, joissa useampi käyttäjä yrittää muokata samaa tietuetta samanaikaisesti. Tällöin eri käyttäjillä on eri versio tietueesta, ja vain käyttäjä, jonka versio vastaa tietokannassa sillä hetkellä olevaa versiota, on oikeutettu tallentamaan.

```
class Role {

    String authority

    static mapping = {
    }

    static constraints = {
        authority blank: false, unique: true
    }
}
```

Kuva 1. Yksinkertainen domain-luokka, jossa on yksi kenttä ja kaksi rajoitusta (constraints).

```
mysql> desc role;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| id        | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| version   | bigint(20)   | NO   |     | NULL    |                |
| authority | varchar(255) | NO   | UNI | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql>
```

Kuva 2. Kuvan 1 domain-luokan pohjalta luodun MySQL-taulun kentät.

Rajoituksilla (constraints) voidaan määrittää erityyppisiä vaatimuksia arvoille. Esimerkkitilanteessa on määritelty, ettei ”authority”-kenttä voi saada tyhjää (blank:false) arvoa. Lisäksi samassa tietokannassa ei voi olla kahta tietuetta, joilla on sama authority-kentän arvo (unique:true). Kuvauksilla (mapping) voidaan mm. poiketa Grailsin sopimuksista ja nimetä tietokantataulut ja kentät oman mieltymyksen mukaan. Kuvan 1 esimerkissä ”String”-tyyppinen muuttuja tallennetaan ”Varchar(255)”-tyyppiseksi tietueeksi. Mikäli muuttujaan haluttaisiin tallentaa pidempiä kuin 255-merkkisiä merkkijonoja, se pitäisi määritellä kuvauksessa (authority type: ”text”), jolloin sen tietueen tyypiksi asetetaan ”Longtext”.

Migraatiot ovat toimintoja, joilla jo olemassa olevan tietokannan rakenteeseen tehdään muutoksia ilman, että vanha data menetetään. Tarve migraatiolle syntyy useimmiten ohjelmaan lisätyistä ominaisuuksista tai asiakkaan muutospyynnöistä. Tähän ongelmaan Grails tarjoaa tehokkaan migraatiotyökalun, joka vertaa olemassa olevan tietokannan sekä projektin domain-luokkien eroavaisuuksia, joiden pohjalta se luo erillisiä migraatioskriptejä, joilla muutokset tehdään tietokantaan. [5.] Migraatioita on syytä tehdä vasta, kun sovelluksesta on julkaistuna jokin pysyvä versio, jonka tietokanta halutaan säilyttää. Kehitysvaiheessa kannattaa käyttää tietokannan luomisessa ”create-drop”-määrettä, joka luo tietokannan uudestaan jokaisen ajon yhteydessä. Tällöin säästytään turhilta migraatioilta, jotka aiheutuvat kehitysvaiheen useista muutoksista.

Tiedonhakuun Grails tarjoaa useita vaihtoehtoja tarpeesta riippuen. Tarkempia hakuja varten GORM tarjoaa ns. ”dynaamisia etsijöitä” (Dynamic finders). Ne ovat ohjelman käännösvaiheen aikana generoituja metodeita, joiden avulla tiedonhaku jonkin kentän arvon perusteella on helpompaa. Monimutkaisimmat hakutulokset voidaan tehdä hyödyntämällä erillistä hakukriteeri (Criteria) oliota, jonka avulla monimutkaisetkin haut on helppo tehdä. Näiden Grailsin tarjoamien hakumenetelmien lisäksi voidaan käyttää myös puhdasta HQL:ää (Hibernate Query Language). Seuraavassa listataan muutamia esimerkkejä yksinkertaisista hakutapauksista, joissa käytetään kuvan 1 domain-luokkaa.

- Role.get(1), palauttaa ”Role”-tyyppisen olion, jonka id on 1.
- Role.list(), palauttaa kaikki tietokannassa olevat tietueet.
- Role.findByAuthority(”ROLE_ADMIN”), esimerkki dynaamisesta etsimestä, joka palauttaa kaikki Role-oliot, joiden ”authority”-kentän arvo on ”ROLE_ADMIN”
- Role.FindByAuthorityLike(”ADMIN”), palauttaa Role-oliot joiden ”authority”-kentän arvo sisältää sanan ”ADMIN”.

- Kriteeri-olioilla kyselyt rakennetaan erilaisista palasista, joiden avulla GORM luo lopullisen mySQL-kyselyn. Kuvassa 3 on esimerkki hieman monimutkaisemmasta kriteeri-oliolla luodusta kyselystä.

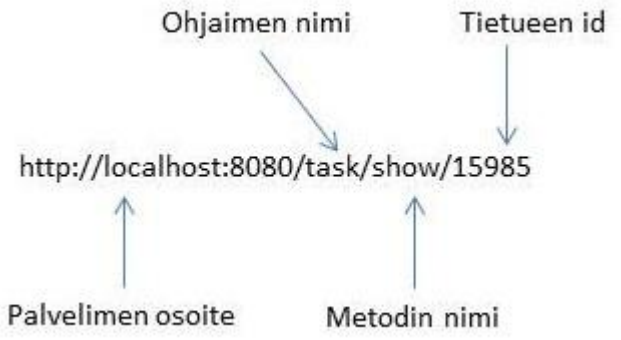
```
def listNotes(Organization organization, params){
    def c = OrganizationItem.createCriteria()
    return c.list(params){
        eq('organization', organization)
        item{
            eq("type",Note.name)
            isNull('customItemType')
            order "lastUpdated", "desc"
            if (params.q) {
                or {
                    like("title", "%" + params.q + "%")
                    like("description", "%" + params.q + "%")
                }
            }
        }
        order "lastUpdated", "desc"
    }
}
```

Kuva 3. Esimerkki Kriteeri-olioiden käytöstä. ”Item” on OrganizationItem-luokan sisältämä relaatio.

3.2.2 Ohjaimet

Ohjaimet eli kontrollerit ohjaavat sovelluksen kulkua. Ne välittävät tietoa tietokantatasolta näkymätasolle ja huolehtivat sivujen vaihtumisesta. Mitä kontrolleria milloinkin käytetään, määräytyy aktiivisen osoitteen perusteella. Aina kun osoite vaihtuu, Grails päättelee sen perusteella, mille kontrollerille pyyntö välitetään. Osoite noudattaa tietynlaista muotoa, jota seuraavaksi tutkitaan esimerkin avulla.

Kuva 4 esittää perinteistä osoiteriviä. Kyseisessä tapauksessa ohjelman nimi on konfiguroitu suoraan palvelimen juureen. Jos näin ei olisi, palvelimen osoitteen ja ohjaimen nimen väliin lisättäisiin ohjelman nimi, esimerkiksi ”localhost:8080/**testisovellus**/task/show/123”. Osoite alkaa palvelimen fyysisellä osoitteella, jonka jälkeen määritetään, mitä kontrolleria kutsutaan: tässä tapauksessa ”task”-nimistä kontrolleria. Kuten domain-luokissa, myös ohjaimissa käytetään Grailsin sopimuksia. Kutsumalla ohjainta ”task” kutsutaan tiedostoa nimeltä TaskController. Kutsumalla ”role” kutsutaan ohjainta RoleController jne. Seuraavaksi määritetään, mitä tuon luokan (esimerkissä TaskController) metodia kutsutaan. Viimeisenä annetaan kutsutulle metodille parametri, esimerkissä tietueen id. Kutsumalla kyseistä osoitetta, tehdään siis käytännössä seuraavanlainen funktiokutsu: ”taskController.show (15985)”.



Kuva 4. Esimerkki Grails-ohjelman osoiterivistä.

Oletetaan ensiksi, että ohjelma on vastuussa sekä tietokannasta että loppukäyttäjän osasta. Tällöin ohjain yleensä tekee tietokantaoperaatioita ja vastaa pyyntöön piirtämällä jonkinlaisen HTML-sivun. HTML-sivun piirtämisestä on lisää luvussa ”Näkymät”. Pilvipalveluiden tapauksessa ohjaimelle lähetetään samanmuotoinen pyyntö, mutta vastauksena palautetaankin yleisimmin xml- tai JSON-muotoista tietoa. Kuvassa 5 on esimerkki normaalin Grails-ohjaimen metodista ja kuvassa 6 API-mallin mukainen metodi.

```
def show(Long id) {
    active_menu = resolveActiveMenu();
    def item = itemService.getOrganizationItem(currentComponent, id)
    if (!item) {
        flash.message = message(code: 'default.not.found.message',
                args: [message(code: 'article.label', default: 'Article'), id])
        redirect(action: "list")
        return
    }
    [item: item]
}
```

Kuva 5. TaskControllerin ”show”-metodi, tietokannasta haetaan parametrina saadun id:n perusteella tietue ja jos tietuetta ei löydy, pyyntö ohjataan ”list” nimiselle metodille. Muussa tapauksessa näkymälle välitetään avain-arvopareja sisältävä muuttuja, jonka sisällä tulos välitetään.

```
@Transactional
def delete(Long id) {
    StageNode.get(id).delete(flush: true)
    render(status: 200, text: 'Delete succesful')
}
```

Kuva 6. Ohjelmointirajapintana käytetyn ohjaimen pelkistetty ”delete”-metodi, joka palauttaa kutsuvalle ohjelmalle tekstin ”Delete successful” sekä statuksen 200 (ok).

Ohjemointirajapinnat eli API:t eroavat sekä toteutukseltaan että käyttötarkoitukseltaan normaaleista Grails-ohjaimista. Niiden tarkoitus on tarjota jotain tiettyä tarkkaan määriteltyä pal-

velua joko ohjelman omaan tai toisten ohjelmien käyttöön. Mikäli sovelluksen backend sekä frontend ovat täysin erilliset toisistaan, niiden välinen kommunikointi toteutetaan useita eri rajapintoja hyödyntäen. Jos backendin toiminnot on toteutettu tarkkaan määriteltyjen rajapintojen avulla, sovelluksen koko frontend voidaan halutessa vaihtaa. Myös täysin erillisten sovellusten välinen tiedonvaihto onnistuu rajapintojen avulla.

Ohjelmointirajapintojen välinen kommunikointi toteutetaan http-verbien avulla (PUT, POST, DELETE, GET). Palvelimella määritellään, miten kuhunkin pyyntöön reagoidaan ja vastataan. Verbien lisäksi kommunikoinnissa hyödynnetään http-tilakoodeja (status codes). Niiden avulla rajapinnat viestittävät kunkin pyynnön sen hetkisen tilan. Esimerkiksi onnistuneesti käsiteltyyn pyyntöön vastataan koodilla 200, virheen aiheuttaneeseen pyyntöön koodilla 500. Luvussa 5 kerrotaan, kuinka rajapintoja voi toteuttaa Grailsin avulla.

3.2.3 URL-reititykset

Mitä tapahtuu, kun sovellus vastaanottaa tietynlaisen pyynnön? Mitä palvelimen pitää mihinkin pyyntöön vastata? Tähän ongelmaan Grails ja muutkin kehykset käyttävät URL-reitityksiä (URL Mappings). Niiden avulla ohjelma osaa välittää tietynmuotoiset pyynnöt oikeille ohjaimille tai välittää aina tietynmuotoiseen pyyntöön tietyn HTML-sivun. Reititykset määritellään UrlMappings.groovy-nimisessä tiedostossa.

Kuvassa 7 on kaksi reititystiedostossa määriteltyä reititystä. Ensimmäinen on ”nimetty reititys” (Named url mapping) ja toinen normaali reititys. Nimettyjä reitityksiä voi käyttää linkkien luonnissa. Nimen avulla Grails osaa generoida oikean osoitteen linkkiin. Muuten nimetyt ja nimettömät toimivat samalla tavalla. ”Action” sekä ”Id” ovat vaihtoehtoisia (?-merkki perässä). Ylemmässä tapauksessa, mikäli osoite alkaa sanalla ”manager”, kaikki pyynnöt ohjataan ohjaimelle nimeltä ManagerController (Grails-sopimus). Eli esimerkiksi pyyntö ”manager/list/45” ohjataan myös ManagerControllerille. Alempi määritys kertoo että kaikki indeksiin tulevat palvelinpyynnöt (”/”) ohjataan ohjaimelle DashboardController, joka tämän sovelluksen tapauksessa vastaa etusivun toiminnoista.

```
name manager: "/manager/$action?/$id?" {
    controller = "manager"
}
"/" {
    controller  = 'dashboard'
}
```

Kuva 7. Kaksi esimerkkireititystä.

3.2.4 Palvelut

Palvelut, eli ”servicet” vastaavat yleiskäyttöisistä toiminnoista sekä sovelluksen logiikasta. Hyvien käytäntöjen mukaan ohjaimet ovat vastuussa vain niille tulevien pyyntöjen ohjaamisesta, kun taas palvelut huolehtivat tietokantakyselyistä ja logiikan suorituksista. Syitä tähän on useita. Ohjelman laajentuessa vastuualueiden eriyttäminen ja koodin uudelleenkäytettävyys nousevat tärkeään rooliin. Palvelut voivat sisältää metodeita, joita kutsutaan useasta eri ohjaimesta, esimerkiksi kuvan viisi ”show”-metodi pitäisi toteuttaa palvelussa, sillä sitä käytetään useasta saman ohjaimen metodista. Sen lisäksi palvelut ovat oletusarvoisesti transaktionaalisia, mikä tarkoittaa, että metodit joko suoritetaan kokonaisina tai ei ollenkaan. Myös esimerkiksi Spring securityn käyttäjäoikeustarkistukset voidaan tehdä ainoastaan palveluiden metodeille.[6.]

Palvelut ovat oletusarvolta ”Singleton”-tyyppisiä, eli koko ohjelman suorituksen ajan jokaisesta palvelusta on olemassa vain yksi instanssi. Tämä edellyttää sen, että palveluiden tulee olla täysin tilattomia, eikä niihin täten tule tallentaa mitään pyyntöihin liittyvää tietoa. Palvelut liitetään ohjaimiin Grailsin Dependency Injection (DI) -toiminnallisuuden avulla. Käytännössä tämä tarkoittaa sitä, että ohjainluokkaan määritellään palvelun niminen muuttuja ja sovelluksen käynnistyessä DI-järjestelmä havaitsee muuttujan ja liittää siihen viittauksen palveluun. Esimerkkisovelluksessa ”ProjectXService” liitetään ohjaimeen ”def projectXService”-määreellä.

3.2.5 Näkymät

Selaimen kutsuessa palvelinta vastauksena on yleensä HTML-tyyppinen tiedosto, jonka selain tulkitsee ja piirtää ruudulle. Palvelin luo kyseisen HTML-tiedoston yhdistämällä ohjaimelta mahdollisesti tulleen tiedon erilliseen näkymätiedostoon (Groovy Server Page). Näkymä on yksinkertaisimmillaan normaalia HTML-kuvausta noudattava tiedosto tai mahdollisesti jokin sivuston osa. Esimerkkiprojektissa loppukäyttäjän osuus on tehty kokonaan Angularilla, joten tarkempi Grailsilla toteutettujen näkymien kuvaus ei kuulu tämän työn aihepiiriin.

3.3 Sovelluksen hallinta

Edellisessä luvussa tutustuttiin Grails-sovelluksen keskeisimpiin toimintoihin. Mallien, ohjainten ja näkymien avulla voi toteuttaa täysverisiä Grails-sovelluksia, mutta vaatimusten kasva-

essa törmää hyvin nopeasti erilaisiin konfigurointitarpeisiin. Seuraavaksi perehdytään, kuinka sovellus räätälöidään omiin tarpeisiin parhaiten soveltuvaksi.

3.3.1 Resurssienhallinta

Resursseja ovat kaikki ydinsovelluksen ulkopuolelta tulevat tiedostot. Tällaisia ovat websovelluksen kohdalla mm. JavaScript-, CSS-, fontti- sekä kuvatiedostot. Resurssien tehokas hallinta on tärkeää, sillä hitailla mobiiliyhteyksillä pienetkin muutokset palvelimelta ladattavan tiedon määrässä vaikuttavat käyttökokemukseen. Toisaalta JavaScript-tiedostojen minifioinnilla saavutetaan myös jonkinlaista tietoturvaa, sillä koodi ei ole selkokielisenä luettavissa selaimessa. Minifioinnilla tarkoitetaan prosessia, jossa lähdekoodista poistetaan kaikki sen suorittamisen kannalta turhat merkit. Tällaisia merkkejä ovat mm. välilyönnit, rivinvaihdot sekä funktioiden parametrien pitkät nimet. Tärkein syy minifiointiin on tiedostokokojen pienentäminen.

Grails 2.4 -versiosta lähtien sovelluksen resurssit käsitellään erillisen ”Asset pipeline” -liitännäisen kautta. [7.] Sen avulla JavaScript- sekä CSS-tiedostot voidaan pakata mahdollisimman pieniksi ja täten nopeuttaa sivun latautumista. Kaikki sovelluksen resurssit sijoitetaan ”assets”-kansion alle, JavaScript-, CSS- sekä kuvatiedostot on hyvä sijoittaa omiin alikansioihinsa. Tiedostoja sisällytetään näkymiin erillisten luettelotiedostojen (manifest) avulla. Luettelot sisällytetään näkymään asset-pipelinen määrettä käyttäen seuraavasti: ”<asset:javascript src="projectX.js"/>. Kuvat 8 ja 9 havainnollistavat, kuinka useita resurssitiedostoja sulautetaan yhdeksi.

projectX.js

```
//= require angular/angular.min
//= require angular-route/angular-route.min
//= require angular-file-upload/angular-file-upload.min
//= require angular-bootstrap/ui-bootstrap-tpls.min
//= require_tree projectX/services
//= require_tree projectX/areas
//= require projectX/main
//= require_tree projectX/directives
```

Kuva 8. ProjectX.js-nimien luettelotiedosto, joka käyttää Asset Pipeline -liitännäisen syntaksia. Lopulta projectX.js-tiedosto sisältää kaikkien luettelossa listattujen tiedostojen sisällöt. Palvelimelle tehdään tämän jälkeen vain yksi pyyntö.

Kuva 9. Palvelimelta pyydetty tiedosto. Tiedosto sisältää kaikkien kahdeksan tiedoston sisällöt minifioituina.[8.]

3.3.2 Riippuvuuksienhallinta

Jokainen Grails-sovellus sisältää vakiona useita liitännäisiä (Plug-in). Vakiona asentuvat ainakin Hibernate- sekä Tomcat-liitännäiset. Erilaiset liitännäiset ovat Grails-kehityksen yksi parhaista puolista, sillä lähes kaikkiin yleisiin käyttötapauksiin löytyy jo erilaisia valmiita liitännäisiä. Valmiiden komponenttien käyttö nopeuttaa sovelluksen kehitystä ja toisaalta vähentää virheiden riskiä, sillä yleensä vähänkään vanhemmat liitännäiset ovat jo hyvin testattuja ja loppuun asti hiottuja. Ihanteellisessa maailmassa kaikki sovelluksen yleiskäyttöiseltä vaikuttavat toiminnot toteutettaisiin liitännäisinä, jolloin niitä voi käyttää myös tulevissa projekteissa. Liitännäisten kehittäminen ei juuri eroa sovelluksen kehittämisestä, mutta on siltikin sen verran suuri aihe, ettei se tämän työn sisältöön mahdu.

Liitännäisten ja yleisten riippuvuuksien hallinta voidaan toteuttaa Grailsin avulla tai hyödyntäen erillistä Maven-työkalua. Maven on yleensä Java-projekteissa käytetty koonnin automatisointityökalu, jolla voidaan mm. kääntää ja julkaista sovelluksia. Mavenia käytettäessä projektin juureen luodaan ”pom.xml”-niminen tiedosto, johon projektin riippuvuudet kuvataan Maven-syntaksin avulla. [9.] Maven tulkitsee kyseisen tiedoston ja sen sisällön perusteella joko poistaa tai asentaa liitännäisiä. Pom-tiedosto luodaan ”create-pom”-komennolla, joka luo tiedoston ja lisää siihen valmiiksi kaikki välttämättömimmät liitännäiset kuten Tomcatin, Hibernaten, mysql-connectorin sekä testaukseen käytettävät työkalut. Maven täytyy asentaa erikseen, eikä se ole osa Grailsia.

3.3.3 Yleinen konfigurointi

Oletusarvoinen Grails-sovellus sisältää neljä eri suoritusympäristöä joita ovat: kehitys, testaus, koonti sekä tuotanto. Sovelluksen käynnistyessä suoritetaan Bootstrap.groovy-niminen alustustiedosto, johon voi määrittää, miten ohjelma halutaan alustaa missäkin ympäristössä. Esimerkiksi kehitysympäristössä voidaan haluta luoda käyttäjä nimeltä admin, jonka salasana on myös admin, mutta tuotannossa vastaavan admin-käyttäjän salasana halutaan asettaa vahvemmaksi. Yleisesti salasanojen sisällyttäminen koodiin on huono idea ja ne tulisi sijoittaa erilliseen konfiguraatiotiedostoon kullekin palvelimelle erikseen. Tarvittaessa sovel-

lukselle voi määrittää myös omia ympäristöjä. Tarve voi syntyä vaikka erilaisten testausympäristöjen myötä.

Toinen tärkeä ympäristökohtainen määrittely suoritetaan Datasource.groovy-nimisessä tiedostossa. Sen avulla määritetään kullekin ympäristölle kaikki tietokantaan liittyvät arvot, kuten tietokannan osoite, tietokannan tunnukset, validointikyselyt yms. Kuvassa 10 on esimerkkisovelluksen kehitysympäristön (development) tietokantamäärittely.

```
environments {
    development {
        dataSource {
            driverClassName = "com.mysql.jdbc.Driver"
            url = "jdbc:mysql://localhost/flow_development?useUnicode=true&amp;characterEncoding=UTF-8"
            username = "flowuser"
            password = "Uid6lEa9vB"
        }
        hibernate {
            show_sql = false
        }
    }
```

Kuva 10. Datasource.groovy. Development (kehitysympäristö) tietokanta-asetukset. Show_sql kertoo Hibernatelle, ettei se tulosta sql-kyselyitä konsoliin. URL on tietokannan osoite, muut kentät ovat itsestäänselviä.

3.3.4 Testaus

Grails sisältää kolmenlaisia testejä: yksikkö-, integraatio sekä toiminnallisuustestejä (Unit, Integration, Functional). Yksikkötestit testaavat nimensä mukaan sovelluksen yksittäisiä pieniä ominaisuuksia. Niillä voidaan testata mm. tietokannan rakennetta tallentamalla olioita ja vertailemalla niiden arvoja. Integraatiotestit eivät juuri muuten poikkea yksikkötesteistä, mutta niitä ajettaessa voidaan hyödyntää Grailsin kontekstia, jota yksikkötestien ajon aikana ei ole. Kontekstilla tarkoitetaan käytännössä sovellusta itseään, eli testien suorituksen aikana sovellus on käynnissä, jolloin testit voivat hyödyntää palveluita, liitännäisiä ja muita ominaisuuksia, mitä normaalin ajon aikanakin voidaan käyttää. Toiminnallisuustesteillä testataan sovelluksen käyttöliittymää. Tähän tarkoitukseen Grailsissä voi käyttää esimerkiksi Seleniumia hyödyntävää liitännäistä nimeltä GEB. Projektissa päädyttiin tekemään integraatiotestit perustoiminnoille. Hyvien periaatteiden mukaan testit tulisi kirjoittaa ennen itse toiminnallisuuden toteuttamista tai vähintäänkin heti sen jälkeen. Useasti kiireestä ja välinpitämättömyydestä testit jäävät kuitenkin kirjoittamatta, mikä kostautuu ennemmin tai myöhemmin. Tämän projektin osalta testit kirjoitettiin prototyyppivaiheen jälkeen, jolloin ominaisuuksia oli vielä melko vähän, ja täten testit oli helpohko tehdä jälkikäteen. Kuvassa 11 on yksi rajapinnan toimintaa testaava yksikkötesti. Testin alussa rajapintaohjaimesta luodaan uusi

instanssi. Tämän jälkeen ohjaimen ”request”, eli pyyntö-muuttujaan asetetaan tyyppi sekä sisältö. Seuraavaksi kutsutaan ohjaimen index-metodia, joka asettaa ohjaimen ”response”, eli vastausmuuttujan arvoksi metodikutsun paluuarvon. Testin lopuksi tarkistetaan että vastausmuuttujan sisältönä on tyhjä lista.

```
void testIndex() {
    def controller = new ProjectXApiController()
    controller.request.contentType = "application/json"
    controller.request.content =
            '{"organization":1"}'.getBytes()
    controller.index()
    assert controller.response.text == '[]'
}
```

Kuva 11. Yksikkötesti, jossa varmistetaan, että tietokannan ollessa tyhjä projektien listaus palauttaa tyhjän listan.

4 Angular

4.1 Yleinen esittely

Angular, tai AngularJS on Googlen kehittämä JavaScript-pohjainen loppukäyttäjän sovelluksiin tarkoitettu ilmainen sovelluskehys. Angular mahdollistaa MVC-arkkitehtuurin hyödyntämisen selainsovelluksissa. Se on kehitetty ns. yksisivu-applikaatioihin (Single Page Application). Nimensä mukaan yksisivu-applikaatiot toimivat siten, että palvelimelta ladataan kaikki tarvittava yhdellä kertaa, jonka jälkeen käyttäjän toimintoja ohjataan vain selaimen avulla ”yhdellä sivulla”. Kutsuja palvelimelle tehdään vain, kun halutaan ladata uutta sisältöä. Tämä ominaisuus tekee Angular-sovelluksista pääasiallisesti hyvin nopeita ja käytettäviä. Tosin suuren datan listauksen tiedetään olevan Angularissa melko tehotonta. MVC-malli mahdollistaa selainsovelluksen kattavan testaamisen, koska sovelluksen vastuualueet voidaan pilkkoa osiin.

Perinteisesti sivustolla jonkin linkin painaminen aiheuttaa pyynnön palvelimelle ja selain jää odottamaan vastauksena uutta sivua. SPA-sovelluksissa näkymien vaihtuminen tapahtuu selaimessa itsessään JavaScriptin avulla. Mikäli sivulla näytetään jotain tietoja, ne haetaan asynkronisesti, ja tulokset piirretään näkyviin pyynnön palattua palvelimelta. SPA-sovellukset soveltuvat hyvin mobiiliympäristöihin, joissa verkkoyhteyksien hitaammat nopeudet huonontavat ”normaaleiden” sivustojen käyttökokemusta.

Kuten Grails-sovelluksessa, myös Angular-sovelluksessa on käytössä mallit (Model), näkymät (View) sekä ohjaimet (Controller). Muita sovellukselle tärkeitä osia ovat palvelut (Service) ja direktiivit (Directive). Seuraavassa osiossa käydään läpi perusteet kustakin Angular-sovelluksen perusosasta.

4.2 Angular-sovelluksen rakenne

4.2.1 Moduulit

Sovelluksen luominen alkaa juurielementin määrittämisellä (kuva 12). Juurielementti on Angular direktiivi ”ng-app”, joka voidaan sijoittaa teoriassa mihin tahansa HTML-dokumentin kohtaan ja samalla HTML-sivulla voi olla useampia Angular-juuria. Hyvien käytäntöjen mukaan, juurielementti olisi syytä sijoittaa joko HTML-sivun <html> tai <body> elementtiin. Moduulit voidaan ajatella erilaisiksi ominaisuuskokoelmiksi, ja juurimoduuli voi koostua useasta eri moduulista. Kaikki uudelleenkäytettävät ohjelman osat on syytä koota omiksi moduuleikseen. Tällaisia ovat mm. direktiivit ja suodattimet (filters). Esimerkkiprojektissa käytetään alue-mallia, jossa jokainen ohjelman näkymä on eritelty omaksi moduuliksi, jotka liitetään yhteen juurimoduulissa. [10.]

```
<body>
<div ng-app="projectX">
    <g:hiddenField name="currentOrganization" id="c-org" value="${currentOrganization.id}"/>
    <div class="row">
        <div ng-controller="appCtrl">
            <ng-view></ng-view>
        </div>
    </div>
</div>

</body>
```

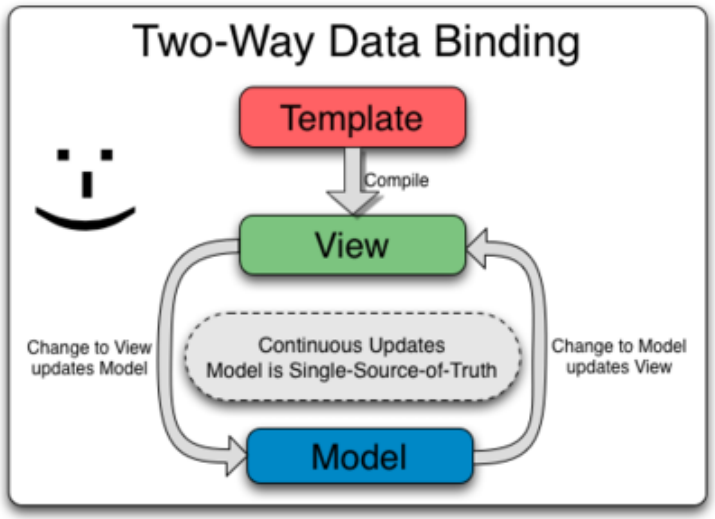
Kuva 12. Body-osa esimerkkiprojektin index.html-sivusta, jossa juurimoduuli on määritetty div-elementtiin, sillä kyseinen HTML-sivu käyttää erillistä layout-tiedostoa, joten tämän sivun body-elementti korvataan kyseisen layout-tiedoston body-elementillä. Tämä on ainoa HTML-sivu, joka palvelimelta ladataan, sen jälkeen näkymiä vaihdellaan Angularin avulla.

4.2.2 Ohjaimet

Ohjaimet ovat Grails-sovelluksen tapaan vastuussa tiedonvälityksestä näkymien ja mallien välillä. Angularin tärkein ominaisuus on näkymien ja mallien välinen ”kahdensuuntainen si-

tominen”, Two-way-binding. [11.] Jokainen ohjain sisältää ”scope”-nimisen muuttujan. Kun scopeen liitetään jokin muuttuja tai funktio, sitä voidaan käyttää näkymissä. Kun scopeen liitettyjen muuttujien arvoja muutetaan joko näkymästä tai ohjaimesta, arvo päivittyy automaattisesti molempiin. Kuva 13 selventää päivityksen kulkua. Kahdensuuntaista sitomista voi helpoiten havainnollistaa yksinkertaisen tekstikentän avulla. Jotta sitominen toimii, tekstikenttä tarvitsee ”ng-model”-määreen, jonka avulla kerrotaan, minkä nimisen scopessa olevan muuttujan arvoa tekstikenttä päivittää. Kun tekstikenttään kirjoittaa, myös ohjaimen scope päivittyy jokaisen kirjaimen kohdalla. Myös esimerkiksi elementtien ”class”-määreen arvot voivat hyödyntää sitomista. Tämän avulla elementin tyyliä voi muokata ohjaimesta käsin, esimerkiksi jos ohjain havaitsee virheellisen syötteen tai käyttäjää on muuten vain hyvä huomauttaa.

Angularin avulla ei tarvita erillistä DOM-puun manipulointia, sillä näkymiä voidaan ohjata muuttujien, direktiivien, suodattimien ja ennen kaikkea reitityksen avulla. Selkeät ohjainlogiikat mahdollistavat selainsovelluksen testaamisen, mikä ei ole useasti mitenkään järkevästi edes mahdollista. Jos kehityksen jossain vaiheessa huomaa työskentelevänsä suoraan DOM-elementtien kanssa, voi olla varma, että sovelluksen rakenteessa on jotain vikaa. Ohjaimet ovat vastuussa sovelluksen sisällöstä ja sen ohjaamisesta. Ohjaimia voidaan käyttää pilvipalveluiden kutsumiseen, mutta yleisesti kaikki kutsut on syytä sijoittaa erillisiin palveluihin. Lisää ohjaimista ja niiden toiminnasta on osiossa viisi.



Kuva 13. Kahdensuuntainen sitominen, nuolet havainnollistavat päivitysten kulkua.

4.2.3 Palvelut

Palvelut tarjoavat nimensä mukaan palveluita yhdelle tai useammalle ohjaimelle, direktiiville, suodattimelle tai toiselle palvelulle. Ne sisältävät yleiskäyttöisiä toimintoja, joita tarvitaan sovelluksen eri osissa. Yleisimmin tällaisia toimintoja ovat pilvipalvelukutsut tai selkeät logiikan laskemiset. Jotta palvelua voidaan kutsua ohjaimesta tai muusta sovelluksen osasta, se pitää liittää ohjaimeen erillisenä riippuvuutena (Dependency). Grailsin tapaan myös Angular-palvelut ovat singletoneja eli kustakin on vain yksi ilmentymä ohjelman ajon aikana.

```
var myServices = angular.module('services.dataService',[]);

myServices.factory("DataService",['$http', function($http) {

    return{
        list: function(org_id){
            return $http(
                {
                    method: 'GET',
                    url: '/projectXEntitys',
                    params:{
                        organization: org_id
                    }
                });
        },
```

Kuva 14. Moduuliin ”services.dataService” luotu angular-palvelu ja sen list-funktio. Tilan säästämisen vuoksi muut funktiot on jätetty kuvasta pois, sillä ne eroavat vain käytettävän verbin ja parametrien suhteen.

Kuvassa 14 on yksi tapa luoda uusi palvelu. Aluksi määritellään uusi moduuli, jolle annetaan nimi sekä mahdolliset riippuvuudet (Dependency) taulukkona. Kuvan tilanteessa moduulin nimeksi tulee ”services.dataServices”, jolla ei ole riippuvuuksia ”[ ]”. Tämän jälkeen moduuliin liitetään erillisen tehdasfunktion avulla uusi ”DataService”-niminen palvelu, joka sisältää riippuvuuden ”$http” palveluun, jota käytetään ajax-kutsujen tekemiseen. List-funktiolla kutsutaan pilvipalvelua, joka palauttaa kaikki parametrina annetun organisaation sisältämät tietueet.

4.2.4 Näkymät

Näkymiä kutsutaan Angular-sovelluksissa templateteiksi tai partiaaleiksi. Ne vastaavat Grails-sovelluksen näkymiä. Ne ovat siis HTML-sivuja, joihin on upotettu Angularin direktiive- jä sekä mallien sidontoja (Data-binding). Angular muodostaa loppukäyttäjälle näkyvän HTML-sivun yhdistämällä templaten sekä malleissa olevan datan. Sovelluksessa on usein yksi ylätason template, jonka sisään piirretään ohjaimesta riippuen erilaisia osa-malleja (par-

tials). Kuvassa 12 oleva HTML-merkkaus esittää tällaista ylätason templatea. Templaten sisällä on Angularin direktiivi ng-view, jonka tilalle piirretään erilaisia osa-templateja riippuen selaimen osoiterivistä, eli siitä ”millä sivulla ollaan”. Yksisivuisuuden johdosta aktiivinen sivu ja aktiivinen kontrolleri määritellään risuaita-syntaksilla, lisää aiheesta reititys-osiossa.

4.2.5 Direktiivit

Direktiivit ovat kuin pieniä rakennuspalikoita, joiden avulla on helppo koota suurempia kokonaisuuksia. Jos sivustolla on elementtejä tai toimintoja, joita tarvitaan useassa eri kohdassa, kyseinen ominaisuus on syytä toteuttaa direktiivinä. Tällaisia ominaisuuksia voi olla mm. erilaiset napit kuten tallenna, muokkaa ja poista. Angular sisältää myös useita valmiita direktiiveja, kuten ng-app, ng-view sekä ng-repeat. Direktiiveillä voi tehdä hyvin yksinkertaisia HTML-sivun osia tai todella monimutkaisia useamman direktiivin yhdistäviä toimintoja.

```
app.directive("editButton",function(){
    return {
        restrict: "E",
        replace: true,
        scope:{
            editAction:'&',
            updateAction:'&',
            object:'='
        },
        templateUrl: "/assets/components/projectX/templates/edit-button.html"
    }

});
```

Kuva 15. Esimerkki ”app”-nimiseen moduuliin lisätystä direktiivistä nimeltä ”editButton”.

Direktiiveille voi määrittää erilaisia arvoja, joista on lisää dokumentaatiossa. [12.] Kuvassa 15 on esimerkki yksinkertaisen direktiivin määrityksestä. Määrittely alkaa ”restrict”-arvolla. Se kertoo, käytetäänkö direktiiviä elementtinä ”restrict: ’E’” (”<edit-button>), vaiko attribuuttina ”restrict:’A’” (<div edit-button>). Molemmissa tapauksissa päädytään samaan lopputulokseen. Hyvien käytäntöjen mukaan direktiivit, jotka luovat jonkin HTML-elementin, tulee määrittää elementteinä, ”E”. Direktiivit jotka lisäävät HTML-elementtiin lisätoimintoja, esimerkiksi hiirikuuntelijan, määritellään atribuutteina: ”A”. ”Replace” kertoo, korvataanko direktiivin sisältämä HTML-elementti, esim ”<div edit-button>”, kyseisen direktiivin templatea käyttäen. Direktiivit on usein syytä toteuttaa siten, että ne eivät pääse vaikuttamaan muihin sivun scope-muuttujien arvoihin. Direktiiville määritelty scope-objekti kertoo, että kyseisellä direktiivillä on

oma eristetty scope eikä se täten vahingossa sekoita ylätason scopejen arvoja. Kuvassa 15 direktiiville annetaan kolme parametria, jotka se liittää omaan eristettyyn scopeensa. Näiden parametrien avulla direktiivi voi vaikuttaa ylätason scopelta parametrina saatuun muuttujaan. Muuttujien perässä olevat merkit kertovat sidonnan tyypistä. ”&”-merkki on yhdensuuntainen sitominen, eli direktiivin muutokset eivät vaikuta ulospäin. ”=”-merkki puolestaan tarkoittaa kahdensuuntaista sitomista, eli arvon muuttuessa muutos näkyy myös ulospäin. Kolmas kuvasta puuttuva sidontamerkki on ”@. Se tarkoittaa, että direktiivi saa kopion muuttujasta eikä sidonta täten toimi kumpaankaan suuntaan.

```
<edit-button edit-action="toggleEdit(object)" update-action="update(object)" object="project"></edit-button>
```

Kuva 16. Esimerkki kuvan 15 direktiivin kutsusta. Lopulliseen HTML-tiedostoon kyseisen kutsun kohdalle piirretään edit-button direktiivin templatena käyttämä edit-button.html-tiedoston sisältö.

4.2.6 Reititys

Reititys toimii käytännössä samalla tavalla Angularissa ja Grailsissä. Reititykset määrittävät, mitä ohjainta ja mitä näkymää milloinkin käytetään. Kun osoiterivi muuttuu, Angular vertaa uutta osoitetta sille määriteltyihin osoitteisiin. Mikäli samanmuotoinen osoite löytyy, kuvassa 12 näkyvän ”<ng-view>”-direktiivin paikalle piirretään määrittelyn mukainen template. Toiminta poikkeaa normaalista web-sovelluksesta siten, ettei uutta sivua tarvitse pyytää palvelimelta. Sivua vaihdellaan ”$location”-nimisen palvelun avulla. Esimerkiksi ”$location.path(”listProjects/”)” siirtää sivun kuvan 18 määrittelyn mukaan projektien indeksi-sivulle.

Grails url	Angular url
localhost:8080/projectX/index	#!/showChecklists/21

Kuva 17. Esimerkkisovelluksen tarkistuslista-sivun osoite, jossa on eroteltuna pysyvä Grails-osoite, ja vaihtuva Angular-osoitteen risuaidalla erotettu palanen.

Kuvassa 17 on eroteltuna Grails-sovelluksen ja Angular-sovelluksen osoitteiden osat. Navigoidessa Angular-sovelluksen sisällä, vain kuvassa 17 oleva ”Angular Url”-osan sisällä oleva osa osoiterivistä muuttuu. Esimerkkiprojektin erikoisesta luonteesta johtuen kyseinen erottelu eroaa normaalista tapauksesta, jossa Angular-osoite on suoraan palvelimen osoitteen perässä. Tämä johtuu siitä, että isäntäsovelluksen näkymistä vain ”projectX/index.gsp” sisältää Angular sovelluksen. Angular versiosta 1.3 lähtien, reititys vaatii erillisen Angular-liitännäisen.

```
app.config(
    [
        '$routeProvider'
        ,'$locationProvider',
    function ($routeProvider, $locationProvider) {
    $locationProvider.hashPrefix('!');
    $routeProvider.
        when('/listProjects', {
            templateUrl: '/assets/components/projectX/areas/projects/index.html'
        }).
        when('/showProjectPhases/:project_id', {
            templateUrl: '/assets/components/projectX/areas/phases/index.html'
        }).
        when('/showChecklists/:phase_id', {
            templateUrl: '/assets/components/projectX/areas/checklists/index.html'
        }).
        when('/', {
            templateUrl: '/assets/components/projectX/areas/projects/index.html'
        })
}]);
```

Kuva 18. Esimerkkisovelluksen juurimoduulin määrittely osioon (config) määritelty reititys.

Jokaisella moduulilla on kaksi osiota: määrittely- sekä ajo-osio. Jokaisen moduulin määrittelyosiot suoritetaan ennen ohjelman varsinaista käynnistämistä. Täten varmistetaan, että kaikki riippuvuudet liitetään ennen kuin niitä käytetään. Kuvassa 18 on esimerkkisovelluksen juurimoduulin määrittelyosio, jossa moduuliin liitetään angular-route.js-tiedoston mukana tullut routeProvider sekä Angularin sisäänrakennettu locationProvider. Tarjoajat tai ”provideerit” toimivat määrittelyvaiheen ajan itse palveluiden sijaisena, ja kun ohjelma ajetaan, niiden avulla päästään käsiksi itse palveluun. Reititys tapahtuu routeProviderin avulla liittämällä yhden tai useampia ”when”-funktioita. Syntaksi on hyvin suoraviivainen: ”kun ollaan sivulla /listProjects”, käytetään määritettyä templatea. Lisäksi määrittelyn yhteyteen voisi määrittää, mitä kontrolleria kussakin tapauksessa käytetään, mutta esimerkkiprojektissa kontrollerit on määritelty kunkin templaten HTML-merkkaukseen erikseen. Kuvassa 19 näytetään, kuinka aktiivinen sivu vaihdetaan $location-palvelun avulla.

```
$scope.showPhases = function(id){
    $location.path('showProjectPhases/' + id );
};
```

Kuva 19. Kontrollerin showPhases funktio, jossa aktiivinen osoite muutetaan muotoon ”/#!/showProjectPhases/123.

Kun kuvan 19 funktiota kutsutaan, routeProvider huomaa, että uusi osoite vastaa sen määrittelyssä olevaa osoitetta. Tämän seurauksena kuvassa 12 näkyvän ”ng-view”-direktiivin pai-

kalle piirretäänkin routeProviderissa määritetyn templaten sisältämä HTML-sivu. Mahdolliset parametrit määritellään reitityksissä kaksoispisteen avulla, esimerkissä objektien ID-kenttä.

4.3 Angularin riippuvuuksienhallinta Grails-ympäristössä.

Edellisessä luvussa käytiin läpi Grails-sovellusten resurssienhallintaa. Siinä ei kuitenkaan otettu kantaa siihen, mistä resurssit ovat alun perin lähtöisin. Resursseilla tarkoitetaan tyyli-, JavaScript-, kuva- sekä fonttitiedostoja. Kutsutaan niitä tästä eteenpäin lyhyemmin selain-resursseiksi. Grailsiin on tehty lukuisia liitännäisiä, jotka sisältävät esimerkiksi Angularin vaatimat JavaScript-tiedostot, jolloin riippuvuudet voi hallita Mavenin avulla samalla tavalla kuin muiden Grails-liitännäisten tapauksessa. Omien kokemusten perusteella kyseinen tapa selainresurssien hallintaan on hyvin hankala ja vaikeasti ylläpidettävä. Parempi tapa on pitää Grails-liitännäiset ja selainresurssit erillä toisistaan.

Kehittäjällä on yleensä kaksi vaihtoehtoa selainresurssien hallintaan. Joko ne ladataan erillisestä CDN-palvelusta jokaisen pyynnön yhteydessä tai ne asennetaan sovellukseen staattisiksi resursseiksi. Nämä sovelluksen ulkoa tulevat resurssit ovat yleensä kolmannen osapuolen tekemiä valmiita tyylikehyksiä, JavaScript-liitännäisiä, kuten jQueryn eri liitäinnäiset, tai kokonaisia JavaScript-kehyksiä, kuten AngularJS. CDN tarkoittaa hajautettua jakeluverkkoa, joka sisältää useita internetiin hajautettuja palvelimia, joilta eri resursseja voidaan ladata [13]. Hajautus mahdollistaa hyvän saatavuuden ja nopeat latausajat. CDN helpottaa resurssien päivitystä, sillä kehittäjän tarvitsee vain vaihtaa CDN-osoitteen tiedoston versio. Tämä lähestymistapa soveltuu parhaiten pieniin sovelluksiin ja demoihin, mutta oikeiden sovellusten tapauksessa kaikki resurssit on syytä säilyttää staattisina palvelimella. Staattisten tiedostojen versioiden ylläpitämisessä on kuitenkin enemmän töitä, kun uudet tiedostot pitää ladata ja päivittää yksi kerrallaan.

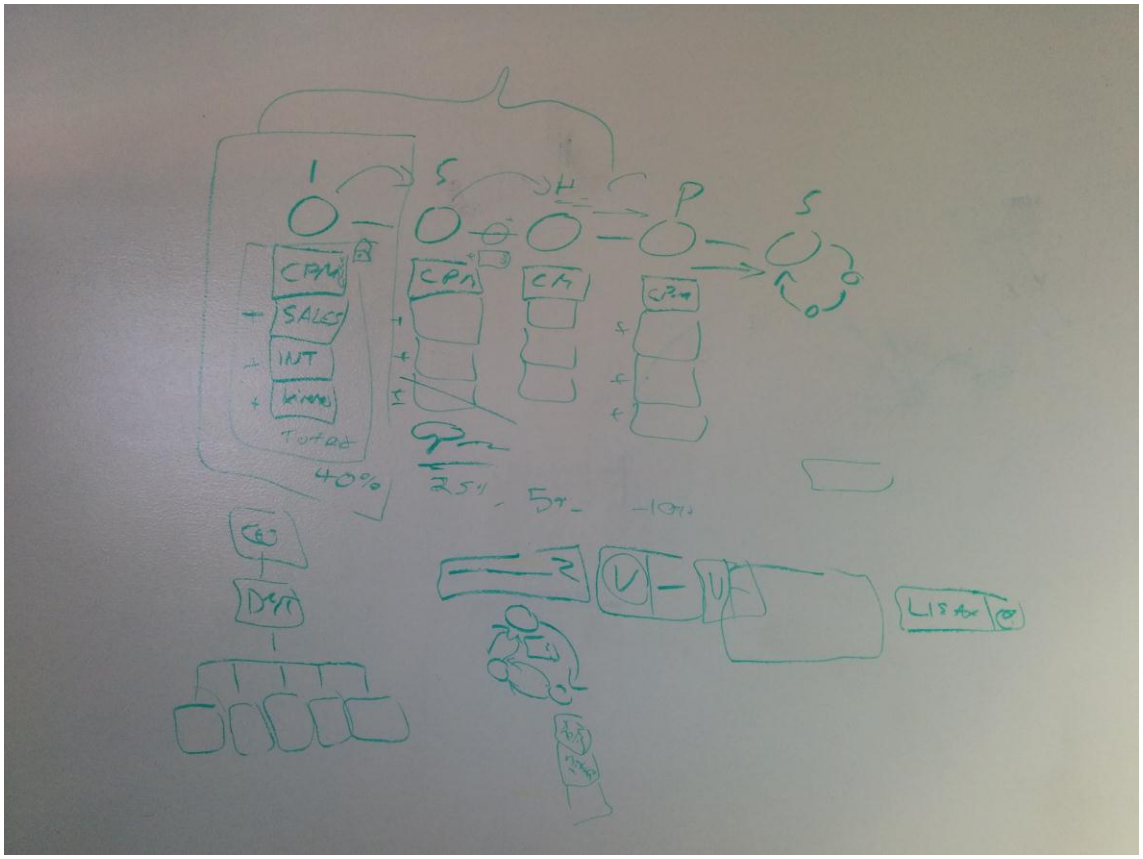
Onneksi staattisten resurssien hallintaan löytyy hyviä työkaluja, joita voi hyödyntää myös Grails-sovelluksissa. Esimerkkiprojektissa kaikkia selainresursseja ylläpidetään Bower-nimisen pakkaushallintatyökalun avulla. Bower asennetaan esimerkiksi Node.js:n mukana tulevan ”Node package managerin” avulla. Node.js asennetaan Ubuntu-ympäristössä apt-get-ohjelman avulla ”apt-get install nodejs”-kommennolla. Node.js:n tarkempi kuvaaminen ei mahdu tämän työn aihepiiriin. Noden asennuksen jälkeen Bower asennetaan komennolla ”npm install -g Bower”. Kun Bower on asennettu, siirrytään Grailsin assets-kansioon, jossa JavaScript-liitännäisten asennus suoritetaan. Esimerkiksi Angular asennetaan seuraavasti: ”bower install angular”. Tämän jälkeen assets-kansioon ilmestyy ”bower_components” -

kansio, jonne asennetut tiedostot siirtyvät. Asset pipeline -liitännäinen (kappaleessa 3.3.1) toimii siten, että se etsii tiedostoja ”assets”-kansion alta ohittaen ensimmäisen kansiotason. Eli kun esimerkkiprojektin luettelotiedostossa (kuva 8) haetaan ”angular/angular-min”, asset pipeline ei huomioi ”bower_components” -kansiota, vaan se etsii suoraan sen alta kyseistä tiedostoa. Angularin tai minkä tahansa muun Bowerin avulla hallinnoidun liitännäisen päivittäminen onnistuu komennolla ”bower update {resurssin nimi}”.

5 Esimerkkisovelluksen toteutus

5.1 Prototyypin määrittely

Projektin idea syntyi työpaikalla oman osastoni johdon tarpeesta. Aluksi oli vain korkeantason ajatus siitä, mitä tuotteella tulisi voida tehdä. Muutaman viikon pohdinnan jälkeen syntyi kuvassa 20 näkyvä määrittely, ja vaatimuksista syntyi ensimmäinen raaka hahmotelma. Tavoitteena on toteuttaa sovellus, jolla projektien eri vaiheita voidaan seurata. Sen avulla kukin projektiin osallistuva kirjaa ylös, mitä on tehnyt milloinkin ja mitä on mahdollisesti jäänyt tekemättä. Projektin aikana järjestettävissä tilannekokouksissa voidaan tarkastella projektin etenemistä ja sovelluksen avulla varmistua siitä, että kaikki tarvittavat toimenpiteet on suoritettu. Jokainen käyttäjä saa sovellukseen henkilökohtaisen käyttäjätunnuksen, jonka avulla hän kirjaa tarkistuslistan kohtia tehdyiksi ja täten allekirjoittaa ne omalla tunnuksellaan. Järjestelmään on myöhemmin tarkoitus tallentaa kaikki projektin tärkeät dokumentit, jotta ne ovat kaikkien osallistujien nähtävissä.



Kuva 20. Projektin alkuperäinen ”määrittelydokumentti”, tussitaululle kiireessä piirretty hahmotelma. Kuvattuna esimerkkiprojekti, jossa on vaiheita (I, S, H, P, S) sekä niissä eri määrä erinimisiä tarkistuslistoja.

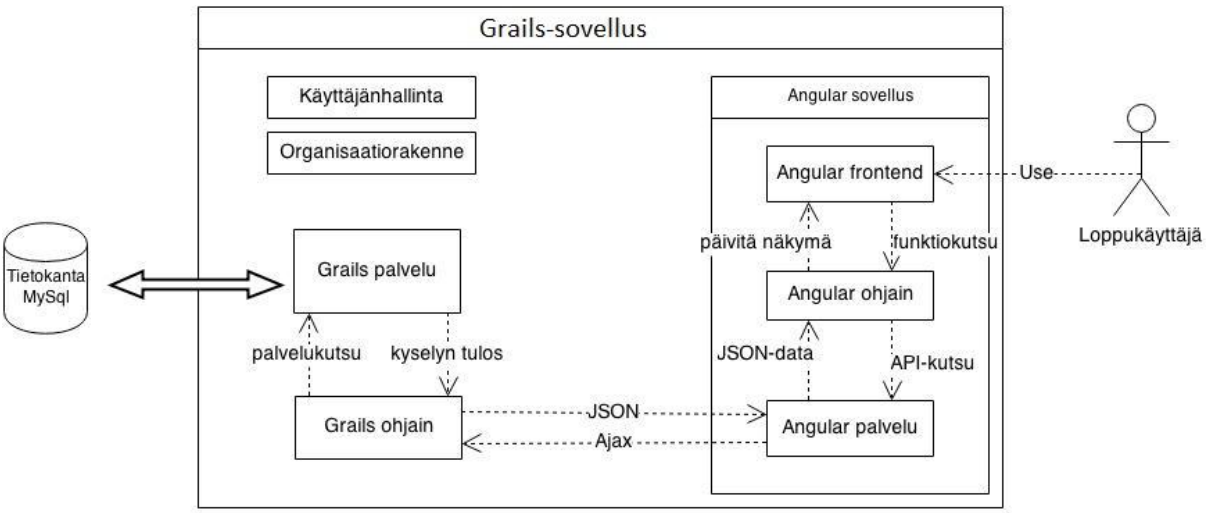
Lopullinen prototyyppivaiheen määrittely muodostui jotakuinkin seuraavanlaiseksi.

- Ohjelmaan voi lisätä projekteja, projekteihin vaiheita, vaiheisiin tarkistuslistoja ja tarkistuslistoihin tehtäviä.
- Käyttäjän tulee voida lisätä, poistaa ja muokata yllämainittuja kohteita.
- Tarkistuslistojen tehtäviä tulee voida merkata tehdyksi sekä kommentoida.
- Jokaisen vaiheen (projekti, vaihe, tarkistuslista) voi tallentaa pohjaksi ja käyttää uudelleen muissa projekteissa.
- Projektit tulee voida liittää eri organisaatiotasoihin, jolloin ne näkyvät vain kyseisessä organisaatiossa.

5.2 Toteutuksen tekninen kuvaus

Sovellus voitaisiin toteuttaa kahtena täysin erillisenä järjestelmänä, jossa Grails toimisi puhtaasti pilvipalveluna ja Angulari:lla tehtäisiin vain pilvipalvelua käyttävä frontend-sovellus. Edellä mainittu lienee se yleisempi lähtökohta näiden kahden sovelluskehyksen yhteistyössä, mutta tämän projektin kohdalla päädyttiin eri ratkaisuun. Projekti toteutetaan jo valmiina olevaan Grails-sovellukseen uutena osana. Kuvassa 21 on havainnollistettu sovellusten sisäkkäisyyttä teknisellä tasolla sekä kuvattu sen toimintaperiaate. Kuvassa 22 on vastaavanlainen erottelu, mutta käyttöliittymän näkökulmasta. Kyseinen ”isäntäsovellus” tarjoaa valmiiksi käyttäjänhallinnan sekä organisaatiorakenteen vähentäen Angular-sovelluksen kompleksisuutta huomattavasti. Järjestelmässä on siis käytännössä kaksi sisäkkäistä sovellusta, jotka kommunikoivat keskenään.

Liitteessä 1 on kuva projektin kansiorakenteesta Intellij IDEA -editorissa. Kuvasta voidaan nähdä, kuinka kaikki Grails-tiedostot on jaoteltu kunkin tiedostotyypin mukaan omiin hakemistoihinsa. Koko Angular-sovelluksen hierarkia on ”assets”-kansion sisällä, jota Grails käyttää mm. JavaScript- ja tyylitiedostojen hallintaan. ”ProjectX.js”-tiedosto on kuvassa 8 (s.11) näkyvä luettelo, jonka avulla koko Angular-sovellus voidaan ladata mille tahansa sivulle.

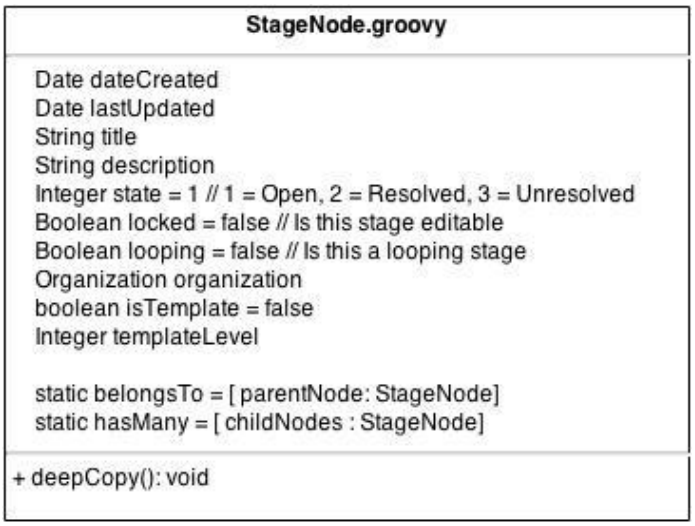


Kuva 21. Sovelluksen arkkitehtuurikuvaus.

Kuva 22. Sovelluksen käyttöliittymä, jossa kokonaisuuden osat ovat eroteltu sinisellä ja keltaisella laatikolla sekä otsikoilla. Osien erottelut on lisätty kuvaan, ne eivät siis näy sovelluksessa.

5.3 Tietokantamalli

Tietokantamallin suunnittelu keskittyi määrittelyvaiheessa esiintyneisiin neljään selkeään elementtiin, jotka ovat projekti, vaihe, tarkistuslista sekä tehtävä. Tutkittaessa kyseisten tekijöiden ominaisuuksia huomattiin, että ne eivät juuri poikkea toisistaan. Projektilla voi olla monta vaihetta, vaiheella monta tarkistuslistaa, tarkistuslistalla monta tehtävää. Jokaiselle elementille tarvitaan samat arvot: otsikko, kuvaus, tila sekä lukittu-tila. Malli toteutettiin käyttäen komposiitti-suunnittelumallia [14], jonka avulla erilaisia puurakenteita on tehokas tehdä. Pohjimmainen ajatus on, että oliolla voi olla monta samantyyppistä lapsioliota, mutta vain yksi vanhempi. Lopputuloksena sovelluksen tietokantamalli saatiin puristettua vain yhdeksi luokaksi ilman, että ominaisuuksista jouduttiin tinkimään. Kuvassa 23 on tietokantamallin tekninen kuvaus.



Kuva 23. Projektissa käytetyn tietokantamallin tekninen kuvaus.

Aikaleimojen päivittämiseen käytetään Grailsin automaattisesti ylläpitämiä ”dateCreated” sekä ”lastUpdated”-kenttiä. Organization-kenttä on isäntäsovelluksen tarjoama toinen PO-GO-luokka. Puurakenne toteutetaan GORM:n avulla ”belongsTo”- sekä ”hasMany”-määrittelyjen avulla. GORM huolehtii oikeanlaisten tietokantataulujen muodostamisesta. Luokkiin voi myös määritellä apufunktioita, jotka suorittavat kyseiselle Domain-luokalle yleiskäyttöisiä toimintoja. Määrittelyn mukaan kukin elementti voidaan tallentaa pohjaksi uudelle elementille, tämä toteutetaan ”deepCopy”-funktion avulla.

5.4 Grails-API:n toteuttaminen

Rajapinnan tai pilvipalvelun toteuttaminen Grailsin avulla on helppoa. Yksinkertaisen palvelun toteuttaminen vaatii vain yhden domain-luokan, joka toimii palvelun resurssina. Tämä kyseinen luokka merkataan Grailsin annotaatioilla (@-merkintä) resurssiksi, jonka jälkeen Grails generoi tarvittavan ohjaimen ja sen funktiot. Tämänlainen lähestymistapa riittää vain hyvin yksinkertaisiin sovelluksiin, eikä sillä päästä projektin vaatimuksiin. Jotta palvelusta saataisiin tarpeeksi joustava ja monipuolinen projektin tarpeisiin, funktiot ja reititykset täytyy kirjoittaa itse. Rajapinnan toteutus aloitettiin luomalla uusi ohjain, projectXAPIController. Kyseinen ohjain on vastuussa rajapintaan tehdyistä normaaleista HTTP-kutsuista, jotka ovat: index, show, create, edit, save, update sekä delete. Näiden lisäksi tarvitaan myös toiminnot kommenttien sekä liitteiden lisäämiselle, projektipohjien luomiselle sekä projektipohjien listaamiselle.

Jotta Grails osaa ohjata tulevat API-kutsut oikealle ohjaimelle sekä funktiolle, tarvitaan oikeanlaiset reitit ykset. Reitit ykset toimivat hyvin samalla tavalla kuin Angularin tapauksessa: reitityksiä ylläpidetään erillisessä UrlMappings.groovy-nimisessä tiedostossa. Reitityksien avulla voidaan esimerkiksi määritellä, mikä ohjain on vastuussa tietyn kutsun käsittelystä tai mikä näkymä piirretään aina tietynlaisen kutsun vastauksena. Pilvipalvelu pyrittiin toteuttamaan REST-mallia noudattaen. [15.] Lyhyesti REST on lista ominaisuuksia ja periaatteita, joita pilvipalvelun tulisi noudattaa.

```
"/projectXEntity"(resources:'projectXApi')
"/projectXEntity/$id?/$action?"(controller: "projectXApi")
"/projectXEntity/$action?"(controller: "projectXApi")
```

Kuva 24. Pilvipalvelun reititysten määrittelyt, UrlMappings.groovy-tiedostosta.

Mikäli tarve olisi vain perustoiminnoille, kuvassa 24 näkyvä ensimmäinen määritys riittäisi hyvin. Se luo reitit ykset kaikille aiemmin mainituille HTTP-verbeille. Kommentoinnin ja tiedostojen liittämisen takia tarvitaan myös kaksi muuta reititystä, joista ensimmäistä käytetään kommentointiin. Kutsumalla ”localhost:8080/projectXEntity/12/comment, reititys ohjaa kutsun projectXApi ohjaimelle ja sen ”comment” nimiselle funktiolle. Viimeinen määritys on pohjien hakemista ja mahdollisten tulevien toimintojen ohjaamista varten. Mikäli erilaisia toimintoja tulee tulevaisuudessa useita, ne on ehkä syytä eristää omiksi palveluikseen, jotta rajapinnat pysyvät selkeinä. Tässäkin tapauksessa kommentointi olisi syytä toteuttaa erillisen Comment-api:n kautta. Kuvan 25 taulukossa on esitettynä koko projektin rajapinnan palvelut esimerkkikutsujen avulla.

http pyyntö (palvelimen osoite poistettu)	http-metodi	funktio	palvelun kuvaus
/projectXEntity/?organization=2	GET	index	Listaa kaikki tietueet, jotka kuuluvat organisaatioon, jonka id on 2
/projectXEntity/1	GET	show	Hakee tietueen, jonka id on 1
/projectXEntity/1	PUT	update	Päivittää tietueen id:llä 1
/http://localhost:8080/projectXEntity/1	DELETE	delete	Poistaa tietueen, jonka id on 1
/projectXEntity/	POST	save	Luo uuden tietueen
/projectXEntity/listTemplates?template_lvl=2	POST	listTemplates	Listaa kaikki pohjat, joiden taso vastaa parametrina annettua tasoa
/projectXEntity/3/createTemplate?templateLevel=2	POST	createTemplate	Luo uuden pohjan tietueesta jonka id on 3, pohjan tasoksi asetetaan 2.
/projectXEntity/11/comment?comment=terve	POST	comment	lisää tietueeseen jonka id on 11, kommentin, jonka sisältö on "terve"
/projectXEntity/upload	POST	upload	Siirtää tiedoston multipart/form-data muodossa palvelimelle

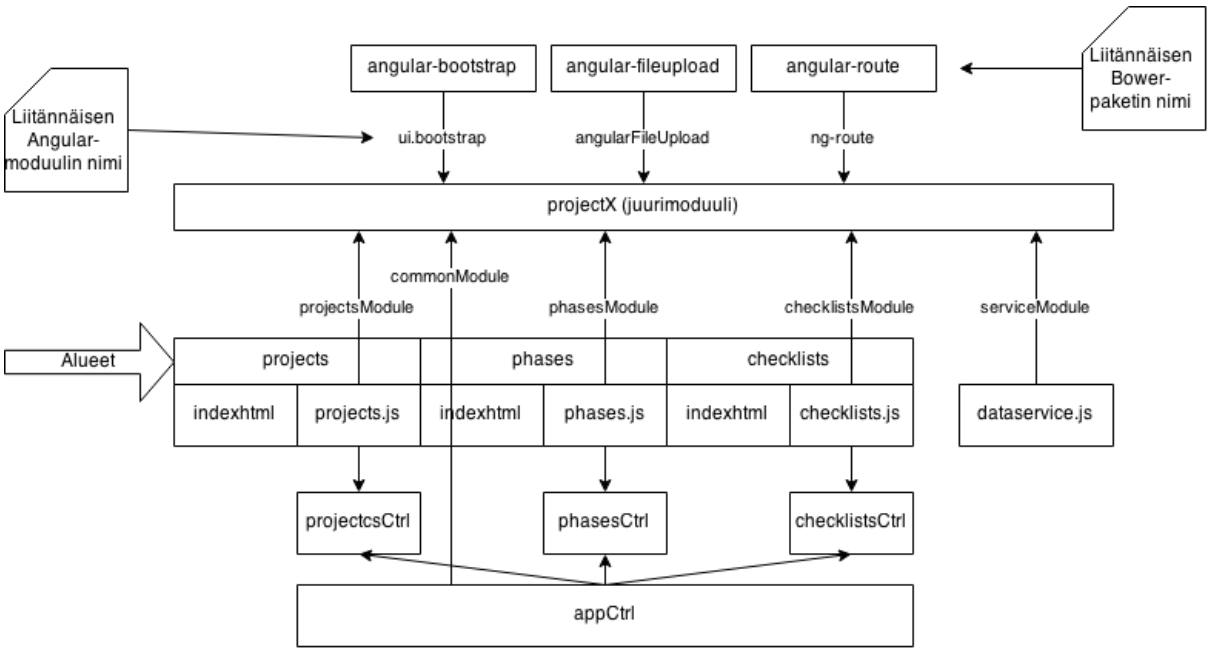
Kuva 25. Projektin API-kuvaus esimerkkikutsujen avulla. Funktiolla tarkoitetaan Grails-ohjaimen funktiota, johon reititykset pyynnön ohjaavat.

Tätä palvelua siis käytetään Angular-sovelluksen dataService-palvelun avulla. Kyseinen arkkitehtuuri mahdollistaisi frontendin irrottamisen kokonaan omaksi sovelluksekseen. Palvelun funktiot palauttavat mahdollisen datan aina JSON-muodossa, mutta esimerkiksi delete-funktio palauttaa vain koodin 200 (ok). Tietokannasta haetun tietueen muuttaminen JSON-muotoon tapahtuu helpoiten käyttämällä Grailsin ”respond”-funktiota ja määrittämällä ohjaimen käyttämään JSON-muotoista vastausta seuraavasti: ”static responseFormats = ['json', 'xml']”. Tämän jälkeen palvelukutsuihin vastaaminen onnistuu esimerkiksi: “respond StageNode.get(id)”, jonka tuloksena kutsuva ohjelma saa JSON-muotoisen esityksen kyseisestä tietokantamallista.

5.5 Angular-frontendin toteuttaminen

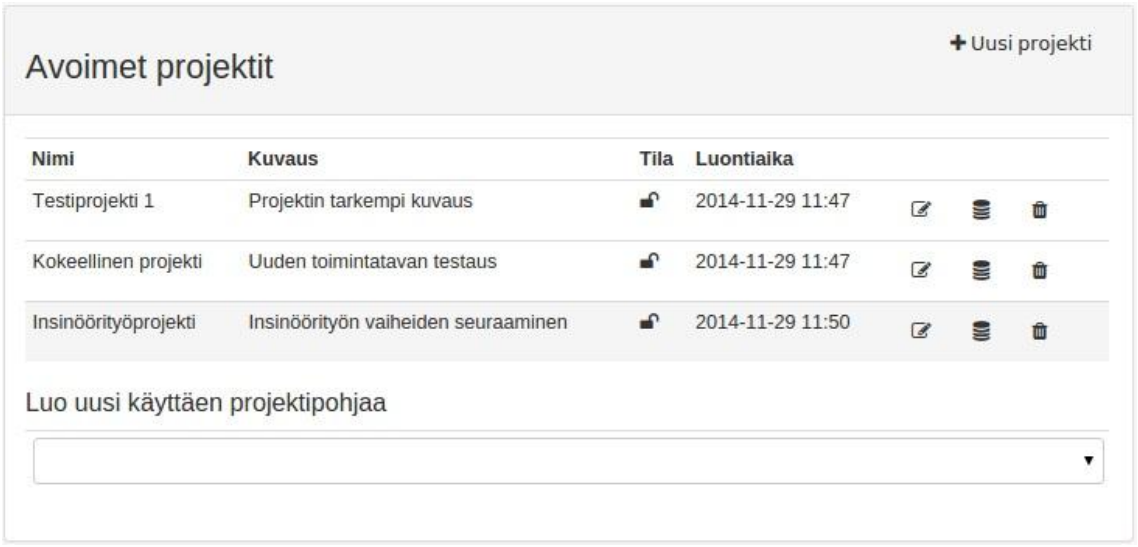
Käyttöliittymä koostuu kolmesta eri näkymästä: projekti-, vaihe- ja tarkistuslistasivusta. Jokainen näkymä on toteutettu omana moduulinaan. Tarkoituksena on eristää jokainen sivu omaksi ”areaksi” (alue). Tämän mallin avulla järjestelmästä saadaan hyvin modulaarinen, kun uusia sivuja tai ”areoita” voidaan liittää vain lisäämällä niitä sovelluksen juurimoduuliin. Jokainen näistä moduuleista tai alueista sisältää index.html-tiedoston sekä alueen nimisen JavaScript-tiedoston. Indeksitiedostossa on sivun sisältö, jossa myös käytettävä ohjain mää-

ritellään. JavaScript-tiedostossa on kyseisen alueen moduulin ja ohjaimen määrittelyt. Mikäli sovellukseen haluttaisi lisätä uusi alue, luodaan vain uusi index.html ja ohjaintiedosto, jonka jälkeen kyseinen moduuli liitetään juurimoduuliin ja sille tehdään uusi reititys.

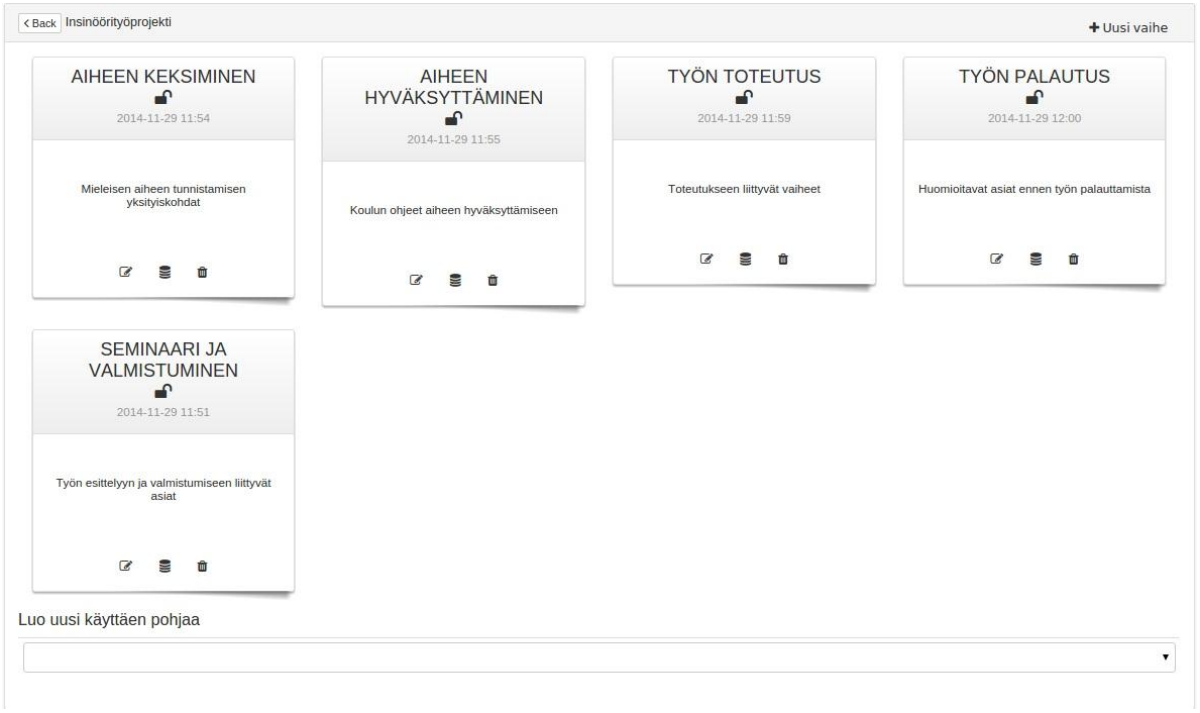


Kuva 26. Frontendin korkean tason arkkitehtuurikuvaus.

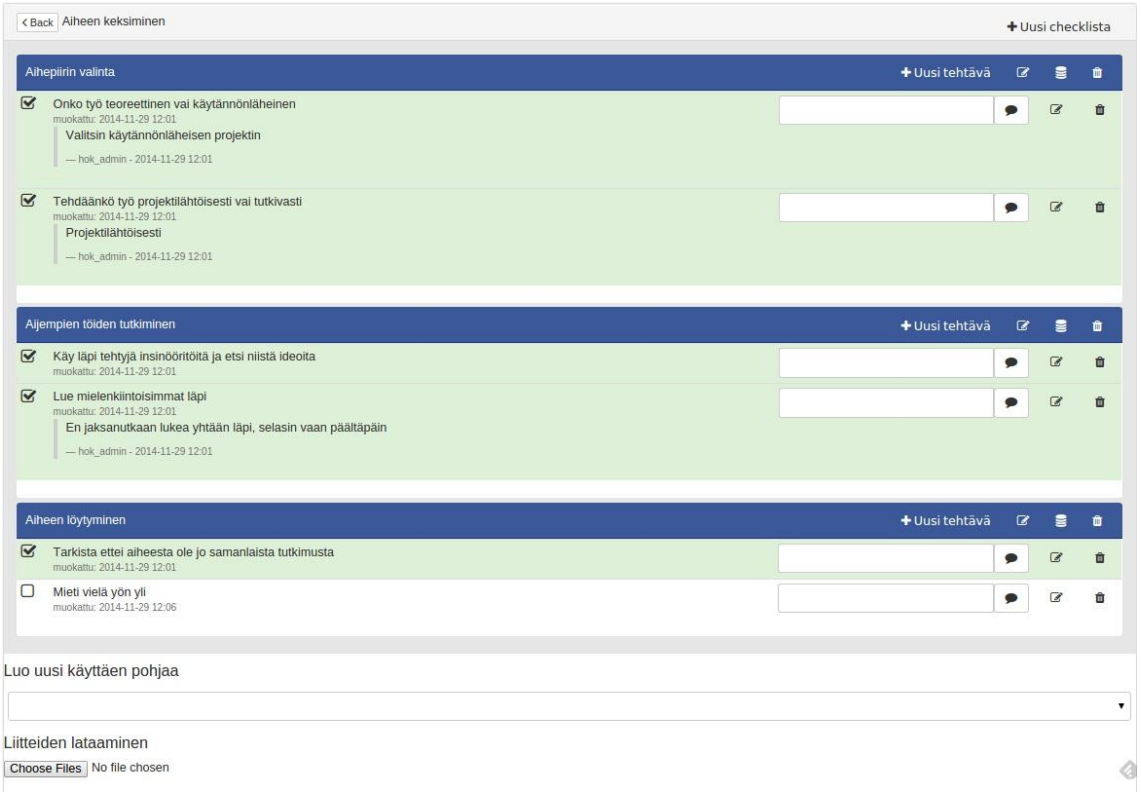
Kuva 26 kuvaa frontend-sovelluksen rakennetta. Projectx, on juurimoduuli, johon kaikki muut osamoduulit liitetään, nuolet kuvaavat liitoksia ja niissä näkyvät tekstit kyseisen moduulin nimen. Kuvaan on lisäksi eroteltu eri sivujen alueet omiksi kokonaisuuksiksi, alue ”projects” on sivuston aloitussivu, josta pääsee ”phases”-sivulle, josta päästään lopulta ”checklists”-sivulle. Navigointi tapahtuu nimenomaan kyseisessä järjestyksessä eteen ja taaksepäin, eli projektisivulta ei voi suoraa päätyä tarkistuslistasivulle. Sivukohtaisten ohjainten lisäksi sovelluksessa käytetään yhtä ylätason ohjainta. Angularissa on siis mahdollista sijoittaa ohjaimia sisäkkäin. Tämä mahdollistaa sen, että alemman tason ohjaimesta voidaan kutsua ylemmän tason ohjaimen funktioita. Ylätason ohjain ”appCtrl” vastaa jokaisella sivulla olevista yleisistä toiminnoista. Tällä hetkellä sen avulla voidaan poistaa tietueita, ladata tiedostoja sekä siirtyä edelliselle sivulle (vastaava kuin selaimen ”edellinen”-painike). Omien moduulien lisäksi sovelluksessa käytetään kolmea liitännäistä: ng-route vastaa reitityksistä, angularFileUpload tiedostojen lataamisesta ja ui.bootstrap erilaisten ponnahdusikkunoiden luonnista.



Kuva 27. Etusivu, ”projects”-alue.

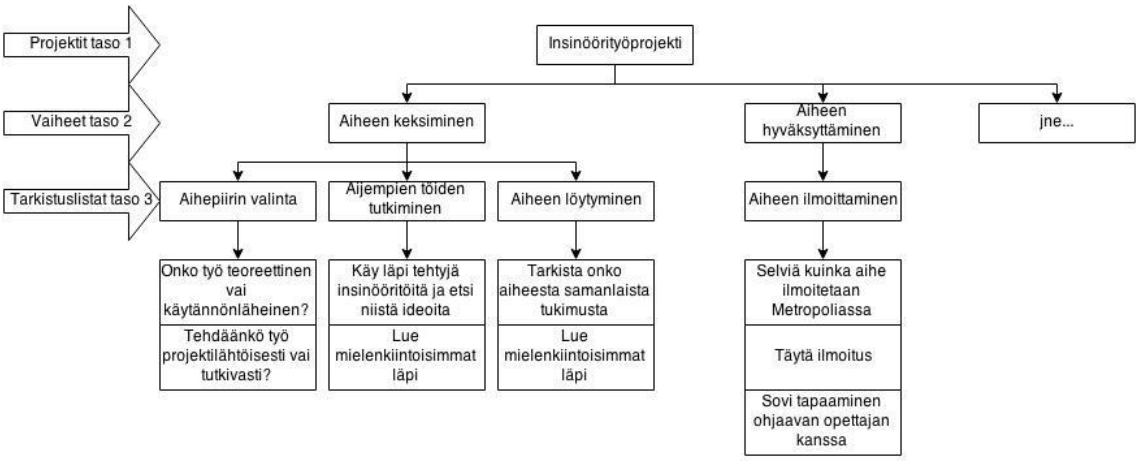


Kuva 28. ”Phases”-alue, jossa kuvattuna projektin ”insinöörityöprojekti” eri vaiheet selityksineen.



Kuva 29. ”Checklists”-alue, jossa vaiheen ”aiheen keksiminen” eri tarkistuslistat tehtävineen ja kommentteineen.

Kuvassa 27 on sovelluksen etusivu, jossa listataan kaikki olemassa olevat projektit, niiden kuvaukset, tilat sekä luontiajat. Oikeassa reunassa olevat napit vasemmalta oikealle: muokkaa, tallenna pohjaksi sekä poista. Muistellaan että tietokantamalli oli toteutettu käyttäen vain yhtä luokkaa, kuvista 28 ja 29 voidaan huomata, että jokaisella elementillä (projekti, vaihe, tarkistuslista, tehtävä) on samat tiedot. Ainoastaan tapa jolla tieto esitetään, on hieman erilainen. Esimerkiksi tehtävissä ja tarkistuslistoissa ”kuvaus”-tietoa ei esitetä lainkaan. Kaikkien elementtien ollessa tietokantatasolla samanlaisia, ne täytyy erotella toisistaan jollain toisella tavalla, johon perehdytään seuraavaksi.



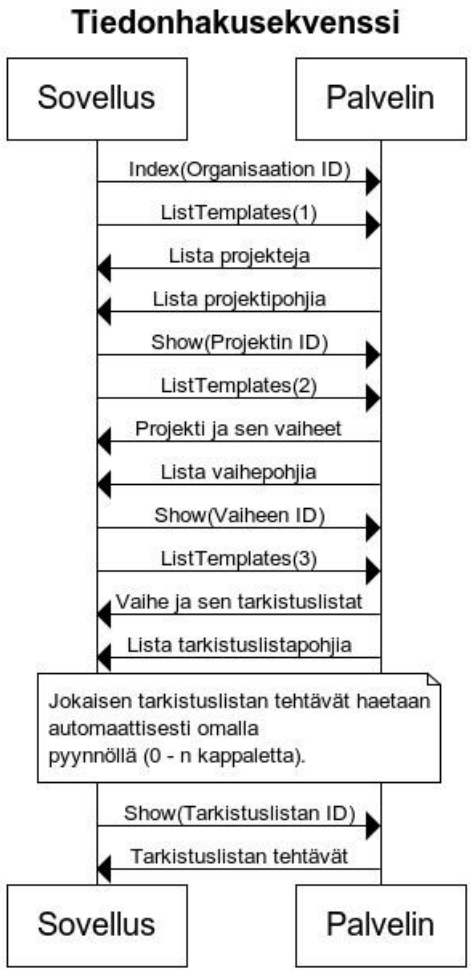
Kuva 30. Sovelluksen elementtien puurakenne ja tasot.

Kuvan 23 tietokantamallista nähdään, että jokaisella ”StageNode”-tietueella on yksi vanhempi ”parentNode” sekä useita lapsia ”childNodes”. Kuva 30 havainnollistaa tätä suhdetta. Puurakenne mahdollistaa käytännössä rajoittamattoman kokoisen hierarkian toteuttamisen. Sovelluksen tapauksessa se tarkoittaa, että projektilla voi olla ääretön määrä vaiheita, vaiheella ääretön määrä tarkistuslistoja ja tarkistuslistalla ääretön määrä tehtäviä. Puun syvyys on projektin vaatimusten takia rajoitettu neljään. Mikäli tulevaisuudessa tehtäville tulisi voida antaa tarkempia yksityiskohtia, puuta voi helposti syventää. Tietojen hakeminen tietokannasta tapahtuu seuraavasti. Sovelluksen alussa listataan kaikki isäntäsovelluksessa valittuun organisaatioon kuuluvat projektit. Kun valitaan jokin projekti, palvelimelle lähetetään rajapintakuvauksen mukainen ”GET”-pyyntö, jonka parametrina on valitun projektin ID. Vastauksena saadaan aina tietue, jonka ID vastaa parametrin ID:tä, ja tämän lisäksi kaikki kyseisen tietueen välittömät lapset. Projektin tapauksessa vastauksena saataisi siis projekti itse ja kyseisen projektin vaiheet. Jos klikataan vaihetta, saadaan vastauksena kyseinen vaihe ja sen tarkistuslistat. Tarkistuslistan tapauksessa vastauksena saadaan tarkistuslista sekä sen tehtävät. Hierarkian lisäksi jokainen taso, josta voidaan luoda pohjia (kirjoitushetkellä tehtäviä ei voi tallentaa pohjiksi), sisältää myös tason numeron. Tasoa käytetään pohjien listaamisessa, esimerkiksi kun vaihe tallennetaan pohjaksi, kyseisen pohjan tasoksi asetetaan ”2”. Myöhemmin, kun palataan vaihesivulle, sivun alalaidassa olevaan pudotusvalikkoon listataan kaikki vaihepohjat, joiden taso on kaksi. Tason numero on kovakoodattuna eri alueiden ohjaimiin.

Kuvassa 31 on pyritty selventämään palvelinpyyntöjen kulkua. Sekvenssi kertoo, mitä pyyntöjä palvelimelle tehdään ja mitä palvelin kussakin tapauksessa palauttaa. Etusivulla kutustaan indeksiä sekä haetaan pohjat, joiden taso on 1, eli tälle tasolle tulee kaikki projektipohjat. Kun käyttöliittymästä klikataan jokin projekti auki, kutsutaan palvelimen show-metodia

kyseisen projektin id:llä. Tämän seurauksena päästään ”projektin sisään”, jolloin vastauksena saadaan haettu projekti ja sen sisältämät vaiheet. Vaihe-sivulla halutaan listata kaikki vaihepohjat, minkä takia palvelimelta pyydetään pohjia joiden taso on 2. Klikatessa jotakin vaihetta päästään kyseisen vaiheen sisään, ja nähdään sen tarkistuslistat. Tarkistuslistapohjat saadaan pyytämällä kaikki tason 3 pohjat.

Eri tasojen avulla voidaan siis erotella, minkä kohdan (projekti, projektin vaihe, tarkistuslista) pohjista on kyse. Esimerkiksi jos tarkistuslistasta luodaan pohja, sitä ei voida hyödyntää projektipohjana. Vastaavasti pohjaksi tallennettua projektia ei voi käyttää tarkistuslistana. Eri osien pohjiksi tallentaminen mahdollistaa uusien kokonaisuuksien luomisen tehokkaasti. Kuvitellaan, että Projekti B on muuten täysin samanlainen kuin Projekti A, mutta B:n tarkistuslistoista yksi on erilainen. Tässä tapauksessa käyttäjä kopioisi Projekti A:n, ja vaihtaisi sen sisältämän väärän projektivaiheen johonkin toiseen. Tämä toinen vaihe voi olla jokin muu ennalta pohjaksi tallennettu vaihe, tai kokonaan uusi. Järjestelmä on siis hyvin modulaarinen, projektien sekä vaiheiden kokoaminen on helppoa ja suoraviivaista. Projektin alkuvaiheessa tosin ei ole vielä varmuutta, onko näin modulaariselle toiminnalle tarvetta.



Kuva 31. Sovelluksen ja palvelimen välinen kommunikointi navigoidessa etusivulta tarkistuslistasivulle. Huomaa eri pohjien tasot eri vaiheissa ”listTemplates(n)”.

Käyttöliittymän suunnittelussa on panostettu selkeyteen ja käytettävyyteen. Ulkoasu on prototyyppivaiheessa vielä keskeneräinen, mutta toiminnot ovat jotakuinkin lopulliset. Mikään sovelluksen toiminto ei vaadi enempää kuin kaksi hiiren painallusta ja jokainen toiminto on joko nimetty mahdollisimman kuvaavasti, tai niihin on lisätty erillinen vihjelaatikko (tooltip). Suunnittelussa elementtien tietojen päivitys sai erityistä huomiota, tavoitteena oli erillisten muokkaussivujen poistaminen. Kuvassa 32 on esitetty elementin päivityksen kulku. Painamalla ”muokkaa”-painiketta otsikko- ja kuvauskenttä muuttuvat tekstikentiksi ja ”muokkaa”-painike ”tallenna”-painikkeeksi. Elementtien luomisessa käytetään myös päivityksessä käytettävää periaatetta, eli erillistä luontisivujakaan ei ole. Kun luodaan uusi elementti, se ilmestyy heti listan viimeiseksi. Uudet elementit erotetaan niiden otsikon ”NEW” avulla. Sovelluksessa navigointi tehdään aina elementtiä painamalla, projektisivulta vaihesivulle pääsee pai-

namalla projektin nimeä. Kuvan 32 vaiheita kuvaavat laatikot ovat kokonaisuudessaan linkkejä, joita painamalla vaihesivulta pääsee tarkistuslistasivulle.

Kuva 32. Esimerkki elementin päivityksen kulusta.

5.6 Testaus

Testaamisessa hyödynnetään Karma-nimistä testausalustaa. Karma luo erillisen http-palvelimen, jonka avulla testejä voidaan suorittaa valittuja lähdekoodeja vasten. Sen avulla voi ajaa useita eri testauskehyksiä mm. Jasminea, Mochaa tai QUnitia. Angular-kehityksessä tulisi hyödyntää TDD (Test Driven Development)-menetelmää, eli testit kirjoitetaan ennen toteutusta. Karma osaa suorittaa testit aina, kun tiedoston sisältö muuttuu. Eli tehdessä esimerkiksi uutta ohjainta ja testien ollessa valmiina, voi varmistaa ominaisuuden toimimisen, kun testit näyttävät vihreää. Projektin kohdalla tätä menetelmää ei oppimiskäyrän takia hyödynnetty, vaan testaus opeteltiin lopuksi. Kuvassa 33 on yksi sovelluksen projektisivua testaava yksikkötesti. Jasmin-testien on tarkoitus olla hyvin luettavia. Testien nimien on hyvä noudattaa aina muotoa: it(”should do something”). Testissä käytetään ”$httpBackend”-nimistä palvelua, jonka avulla testejä voidaan ajaa ilman oikeaa API:a. Kyseiselle palvelulle määritellään, miten sen tulee vastata kuhunkin pyyntöön. Kuvan 33 testin alussa $httpBackend-palvelu asetetaan palauttamaan testitiedostossa määritelty JSON-objekti nimeltä ”data”, jonka muoto on sama kuin palvelimelta oikeasti tulevalla datalla. Tämän jälkeen kutsutaan ”createController”-nimistä metodia, joka luo instanssin testattavasta ohjaimesta. Tämän jälkeen kutsutaan $httpBackendin flush-metodia, minkä seurauksena testin alussa asetettu data palautetaan ohjaimelle. Lopuksi suoritetaan varsinainen testaaminen. Jasmin käyttää ”expect”-nimistä metodia ehtojen arvioimiseen, sekä sarjan metodeja

varsinaisten muuttujien arvojen tarkasteluun. Kuvassa 33 käytetään mm. ”toEqual”-metodia, mikä varmistaa, että ohjaimen projektilistan ensimmäisen alkion id on 1207.

Testit kannattaa kirjoittaa ennen toteutusta myös siksi, että se pakottaa kehittäjän miettimään logiikan hyvin tarkasti. Mikäli testitapaus on hyvin kompleksinen ja epäselvä, ohjain on todennäköisesti vastuussa liian suuresta toiminnosta. Kuten jo aiemmin todettiin, ohjainten tulisi vastata vain yksinkertaisten arvojen välittämisestä näkymän ja ulkoisen palvelimen välillä. Mikäli testitapaus siis näyttää liian monimutkaiselta, ohjaimen logiikka on syytä pilkkoa pienempiin osiin hyödyntäen erillisiä palveluita (service).

```
it('Should list available projects', function() {
    $httpBackend.expectGET("/projectXEntity").respond(data);
    $httpBackend.expectPOST("/projectXEntity/listTemplates?template_lvl=1").respond({});
    createController();
    $httpBackend.flush();
    expect(scope.projects[0].id).toEqual(1207);
    expect(scope.projects[0].parentNode).toBeNull();
    expect(scope.projects[0].childNodes.length).toEqual(0);
    expect(scope.loading).toBeFalsy();
});
```

Kuva 33. Jasminella tehty yksikkötesti.

5.7 Ylläpito- ja jatkokehitystarpeet

Sovelluksella ei vielä ole selkeää tiekarttaa tulevaisuuden kehitystä varten. Se on vielä varhaisessa pilotointikäytössä. Osaa vaadituista ominaisuuksista ei otettu prototyypin määrittelyssä huomioon, koska ne eivät olleet kriittisiä sovelluksen toimimisen kannalta. Sovellusta kehitetään tulevien ideoiden ja tarpeiden mukaan siten, että sille voidaan löytää käyttötarkoitus myös projektinhallinnan ulkopuolelta.

Kehityksen seuraavassa vaiheessa käyttöliittymäkuvissa näkyvään lukko otetaan oikeasti käyttöön. Sen avulla hallinnoiva käyttävä voi merkitä projektin tai vaiheen lukituksi, jonka jälkeen sen editointi ei ole enää mahdollista. Ulkoasu on vielä prototyypissä melko rujo, joten sen parantaminen on myös tärkeää tulevissa versiossa. Liitteiden lisääminen oli alun suunnitelmissa tärkeänä ominaisuutena, mutta se kuitenkin päätettiin jättää pois. Vaikkakin prototyypissä liitteitä voi ladata, ei niitä voi vielä mistään myöhemmin tarkastella. Mikäli sovellus koetaan sisäisessä käytössä hyödylliseksi, se pitäisi todennäköisesti siirtää jo käytössä olevalle tuotantopalvelimelle tai kokonaan omalle palvelimelleen. Tällä hetkellä palvelu toimii kehityspalvelimella, joka ei ole tarpeeksi stabiili ja tietoturvallinen.

Pidemmän tähtäimen suunnitelmissa on sovelluksen liittäminen olemassa oleviin projektinhallintatyökaluihin: Boseen tai Focukseen. Tavoitteena on Boseen tai Focukseen tehtyjen projektien tuonti rajapinnan kautta. Tämä helpottaisi sovelluksen käyttämistä entisestään, kun projekteja ja niiden tietoja ei tarvitsisi manuaalisesti syöttää. Kaukaisimmissa visioissa sovelluksen on tarkoitus toimia useiden eri tietolähteiden koontisovelluksena, jossa projektin tärkeitä tietoja kerätään eri palveluista yhteen paikkaan. Sovellukseen voisi kerätä mm. projektissa mukana olevien tuntikirjaustietoja ja erilaisia projektin tulosta seuraavia kuvaajia ja laskelmia. Nykyinen toteutus toimisi vain projektin tehtävänseurantana, eli vain yhtenä osana suurempaa projektinseurantakokonaisuutta.

6 Yhteenveto ja tulokset

6.1 Projektin yhteenveto

Tässä työssä perehdyttiin kahteen uudehkoon web-sovelluskehykseen, Grailsiin sekä AngularJS:ään. Tavoitteena oli tarjota perusteet kummastakin teknologiasta ja kertoa, kuinka niitä voidaan käyttää yhdessä. Työn tuloksena syntyi projektinhallintaa helpottava työkalu, jota todennäköisesti käytetään Digia Finland Oy:n projektien apuna. Työn ensimmäisessä kappaleessa käsiteltiin Grailsiä, toisessa Angularia ja kolmannessa keskityttiin niitä soveltavan projektin kuvaamiseen.

Projektin alkaessa kokemus Angularista rajoittui yhteen muutaman tunnin mittaiseen koodaustapahtumaan viime syksynä. Kokemusta Grailsista oli kuitenkin jo lähes vuoden verran, joten siltä osin projekti aloitettiin luotettavin mielin. Varmaa oli, että ainakin jonkinlainen versio sovelluksesta saadaan valmiiksi järkevässä ajassa. Alun perin ideana oli, että osa koodista tehtäisiin työtehtävien ohessa, mutta työkiireiden takia koodaaminen tehtiin lähes kokonaan työajan ulkopuolella.

Myös omalla kohdallani Angularin oppimiskäyrä oli alussa melko jyrkkä. Mutta kovan motivaation, oppimishalun ja työkavereilta saatujen kehujen avulla projekti eteni kovaa vauhtia. Uutta sovelluskehystä opeteltaessa ensimmäinen versio harvoin jää viimeiseksi. Usein jollain tavalla toteutettu ominaisuus osoittautuu myöhemmin opitun perusteella ”vääräksi” tai muuten vaan huonoksi tavaksi toteuttaa kyseinen ominaisuus. Näin kävi usein myös tämän projektin aikana: kansiorakenne, moduulit, näkymät ja ohjaimet kokivat useamman uudelleentoteutuskierroksen. Prototyyppiversion jäädyttämisen jälkeen, tämän työn kirjoittamisen aikana on opittu paljon uusia ominaisuuksia sekä Grailsista, että Angularista, jotka vaatisivat sovellukseen useita muutoksia.

6.2 Projektin tulokset

Projekti simuloi omien kokemuksieni mukaan ohjelmistoprojektia varsin hyvin: tiukka aikataulu (tässä työssä vain itse asetettu), ympäripyöreät määrittelyt ja uusien teknologioiden oppiminen ovat ohjelmoijan arkipäivää. Tavoite oli kuitenkin melko selkeä heti alusta lähtien, ja ajoittaisten suunnittelupalavereiden myötä projektin lopputulos osui jotakuinkin kohdalleen. Tuloksena saatiin alun määrittelyt hyvin kattava ja vähintäänkin hyvällä tasolla toimiva sovel-

lus. Tällaiset ns. ”matalan riskitason” projektit, joita ei tehdä suoraan asiakkaalle, ovat parhaita uusien asioiden oppimisessa. Vapaampi työtahti mahdollistaa monesti laadukkaamman koodin tuottamisen, kun asioihin voi paneutua tarkemmin. Tästä syystä voinkin todeta, että projektin tuloksena ei syntynyt vain uutta työkalua. Tuloksena syntyi myös paljon osaamista, jota voidaan hyödyntää tulevissa ”oikeissa” asiakasprojekteissa.

Lähteet

1 Pivotal kotisivu. Verkkosivu. <http://grails.org/doc/latest/guide/introduction.html> Luettu 20.12.2014.

2 GVM kotisivu. Verkkosivu. <http://gvmtool.net/> Luettu 15.12.2014.

3 GORM-ohje. Verkkodokumentti. <http://grails.org/doc/latest/guide/GORM.html> Luettu 17.11.2014.

4 Convention over configuration. Wikipedia-artikkeli. <http://en.wikipedia.org/wiki/Convention_over_configuration> Luettu 19.11.2014.

5 Grails DBM-ohje. Verkkodokumentti. <http://grails.org/plugin/database-migration> Luettu 24.11.2014.

6 Spring security -manuaali. Verkkodokumentti. <http://grails-plugins.github.io/grails-spring-security-core/> Luettu 24.11.2014.

7 Asset pipeline -manuaali. Verkkodokumentti. <http://grails.org/plugin/asset-pipeline> Luettu 25.11.2014.

8 Minification. Wikipedia-artikkeli. <http://en.wikipedia.org/wiki/Minification_%28programming%29> Luettu 26.11.2014.

9 Maven kotisivu. Verkkosivu. <http://maven.apache.org/> Luettu 30.11.2014.

10 Alue suunnittelumalli. Github projekti. <https://github.com/patrikkeinonen/angularblog> Luettu 10.11.2014.

11 Kahdensuuntainen sitominen. Verkkodokumentti. <https://docs.angularjs.org/tutorial/step_04> Luettu 13.11.2014.

12 Angular direktiivit. Verkkodokumentti. <https://docs.angularjs.org/guide/directive> Luettu 16.11.2014.

13 CDN, Wikipedia-artikkeli. <http://en.wikipedia.org/wiki/Content_delivery_network> Luettu 21.1.2015.

14 Rekursiokooste suunnittelumalli. Wikipedia-artikkeli. <http://en.wikipedia.org/wiki/Composite_pattern> Luettu 1.11.2014.

15 REST-säännöstö. Verkkodokumentti. <http://spring.io/understanding/REST> Luettu 20.12.2014.

Grails-sovelluksen kansiorakenne